DirectX Graphics

This section provides information about using the Microsoft® DirectX® Graphics application programming interfaces (APIs).

As with other components of DirectX, DirectX Graphics can be used with C, C++, and Microsoft Visual Basic®.

For an overview of the organization of the DirectX Graphics Help, see **Error! Bookmark not defined.**.

Roadmap

Information on Microsoft® DirectX® Graphics is presented in the following sections.

What's New in DirectX Graphics. New features and functionality of this component in DirectX 8.0. If you've used Microsoft Direct3D® or Microsoft DirectDraw® before, read this section first, because much has changed since DirectX 7.0.

Introduction to DirectX Graphics. An overview of what DirectX Graphics is and what it can do for your application, together with a look at some key principles and necessary background information.

Understanding DirectX Graphics. A deeper look at the underlying structure. This section won't teach you how to implement vertex lighting or texture mapping in your application, but it will help you understand the mechanics of the API when you get into the details.

Using DirectX Graphics. A guide to using the API. You'll probably want to familiarize yourself with the table of contents for this section and then refer to parts of it as you need specific information. Use it in conjunction with the reference section.

Advanced Topics in DirectX Graphics. Information of interest for creating applications that take full advantage of the power of DirectX Graphics.

Programming Tips and Tools. Miscellaneous information about compiling, debugging, and optimizing DirectX Graphics applications. In addition, this section contains information on issues that may be encountered when using pre-DirectX 8.0 drivers.

[C++]
DirectX Graphics C/C++ Tutorials. Step-by-step implementation of basic functionality. If you learn best by doing, this is a good place to start.

DirectX Graphics C/C++ Samples. A guide to the sample applications in the SDK to point you to the sample code you need.

Direct3D C/C++ Reference. Detailed information for all the API elements declared in the Microsoft Direct3D® header files.

Direct3DX C/C++ Reference. Detailed information for all the API elements declared in the Direct3DX utility library header files.

Direct3DX Shader Assemblers Reference. Detailed reference information on the Direct3DX vertex and pixel shader assemblers.

Effect File Format. Information on the format of effect files, which consist of a set of parameter declarations, followed by descriptions of various techniques

X File C/C++ Reference. Detailed information for all the API elements declared in Dxfile.h.

X File Format Reference. Detailed information on the DirectX (.x) file format.

[Visual Basic]
DirectX Graphics Visual Basic Tutorials. Step-by-step implementation of basic functionality. If you learn best by doing, this is a good place to start.

DirectX Graphics Visual Basic Samples. A guide to the sample applications in the SDK to point you to the sample code you need.

Direct3D Visual Basic Reference. Detailed information on the Direct3D® API elements in the DirectX for Visual Basic type library.

Direct3DX Visual Basic Reference. Detailed information on the Direct3DX API elements in the DirectX for Visual Basic type library.

Direct3DX Shader Assemblers Reference. Detailed reference information on the Direct3DX vertex and pixel shader assemblers.

Effect File Format. Information on the format of effect files, which consist of a set of parameter declarations, followed by descriptions of various techniques

X File Visual Basic Reference. Detailed information on the DirectX (.x) API elements in the DirectX for Visual Basic type library.

X File Format Reference. Detailed information on the .x file format.

What's New in DirectX Graphics

Microsoft® DirectX® 8.0 maintains backward compatibility by exposing and supporting objects and interfaces offered by previous releases of DirectX. However, many new features and performance enhancements have been added as part of the DirectX 8.0 new Microsoft Direct3D® API interfaces.

Complete integration of DirectDraw and Direct3D

Simplifies application initialization and improves data allocation and management performance, which reduces the memory footprint. Also, the

integration of the graphics APIs enable parallel vertex input streams for more flexible rendering.

Programmable vertex processing language
Enables you to write custom shaders for morphing and tweening animation, matrix palette skinning, user-defined lighting models, general environment mapping, procedural geometry, or any other developer-defined algorithm.

Programmable pixel processing language
Enables you to write custom hardware shaders for general texture combining expressions, per-pixel lighting (bump mapping), per-pixel environment mapping for photoreal specular effects, or any other developer-defined algorithm.

Multisampling rendering support
Enables full-scene anti-aliasing and multisampling effects, such as motion blur and depth-of-field.

Point sprites
Enables high-performance rendering of particle systems for sparks, explosions, rain, snow, and so on.

3-D volumetric textures
Enables range-attenuation in per-pixel lighting and volumetric atmospheric effects, and can be applied to more intricate geometry.

Higher-order primitive support
Enhances the appearance of 3-D content and facilitates the mapping of content from major 3-D authoring tools.

Higher-level technologies
Includes 3-D content creation tool plug-ins for export to Direct3D of skinned meshes using a variety of Direct3D techniques, multiresolution level-of-detail (LOD) geometry, and higher-order surface data.

Indexed vertex blending
Extends geometry blending support to allow the matrices used for vertex blending to be referred to using a matrix index.

Expansion of the Direct3DX Utility Library
Contains a wealth of new functionality. The Direct3DX utility library is a helper layer that sits on top of Direct3D to simplify common tasks encountered by 3-D graphics developers. Includes a skinning library, support for working with meshes, and functions to assemble vertex and pixel shaders. Note that the functionality supplied by D3D_OVERLOADS, first introduced with Microsoft DirectX® 5.0, has been moved to the Direct3DX utility library.

Introduction to DirectX Graphics

This section introduces the Microsoft® DirectX® Graphics APIs, providing a high-level overview of technology and services.

- The Power of DirectX Graphics

- Getting Started with DirectX Graphics

The Power of DirectX Graphics

Microsoft® Direct3D® is designed to enable world-class game and interactive three-dimensional (3-D) graphics on a computer running Microsoft Windows®. It is a drawing interface that provides device-dependent access to 3-D video-display hardware in a device-independent manner.

Direct3D is a low-level 3-D API that is ideal for developers who need to port games and other high-performance multimedia applications to the Windows operating system. Direct3D provides a device-independent way for applications to communicate with accelerator hardware at a low level.

Direct3D is a software interface that provides direct access to display devices while maintaining compatibility with the Windows graphics device interface (GDI), providing a device-independent way for games and Windows subsystem software, such as three-dimensional (3-D) graphics packages, to gain access to the features of specific display devices.

Direct3D provides excellent game graphics on computers running Windows 95 or later, or Windows 2000.

Some of the advanced features of Direct3D are:

- Switchable depth buffering (using z-buffers or w-buffers).
- Flat and Gouraud shading.
- Multiple light sources and types.
- Full material and texture support, including mipmapping.
- Robust software emulation drivers.
- Transformation and clipping.
- Hardware independence.
- Full support on Windows 95, Windows 98, Windows ME, and Windows 2000.
- Support for the Intel MMX architecture, Intel Streaming Single-Instruction, Multiple-Data (SIMD) Extensions (SSE)®, and AMD's® 3DNow® architecture.
- Support for a Hardware Abstraction Layer (HAL). This provides a consistent interface through which to work directly with the display hardware, getting maximum performance.
- Support for page flipping with multiple back buffers in full-screen applications. For more information, see Page Flipping and Back Buffering.
- Support for clipping in windowed or full-screen applications.
- Support for 3-D z-buffers.
- Access to image-stretching hardware.
- Simultaneous access to standard and enhanced display-device memory areas.

- Other features, including exclusive hardware access and resolution switching.

These features combine to enable you to write applications that easily outperform standard Windows GDI-based applications and even MS-DOS applications.

The world management of Direct3D is based on vertices, polygons, and commands that control them. It enables immediate access to the transformation, lighting, and rasterization 3-D graphics pipeline. If hardware isn't present to accelerate rendering, Direct3D offers robust software emulation.

Direct3D provides simple and straightforward methods to set up and render a 3-D scene. The key set of rendering methods is referred to as DrawPrimitive methods. They enable applications to render one or more objects in a scene with a single method call. For more information about these methods, see Rendering Primitives.

Direct3D enables a low-overhead connection to 3-D hardware. This comes at a price, however. You must provide explicit calls for transformations and lighting, you must provide all the necessary matrices, and you must determine what kind of hardware is present and its capabilities.

Getting Started with DirectX Graphics

This section provides an overview of graphics programming with Microsoft® Direct3D®. Each concept discussed here begins with a nontechnical overview, followed by specific information about how Direct3D supports it.

You don't need to be a graphics guru to benefit from this overview. In fact, if you are one, you might want to skip this section entirely and move on to the more detailed information in the Using DirectX Graphics section. When you finish reading these topics, you will have a solid understanding of basic Direct3D graphics programming concepts.

The following topics are discussed.

- 3-D Coordinate Systems and Geometry
- Shading Techniques
- Matrices and Transformations
- Vectors, Vertices, and Quaternions
- Copying Surfaces
- Page Flipping and Back Buffering
- Rectangles

3-D Coordinate Systems and Geometry

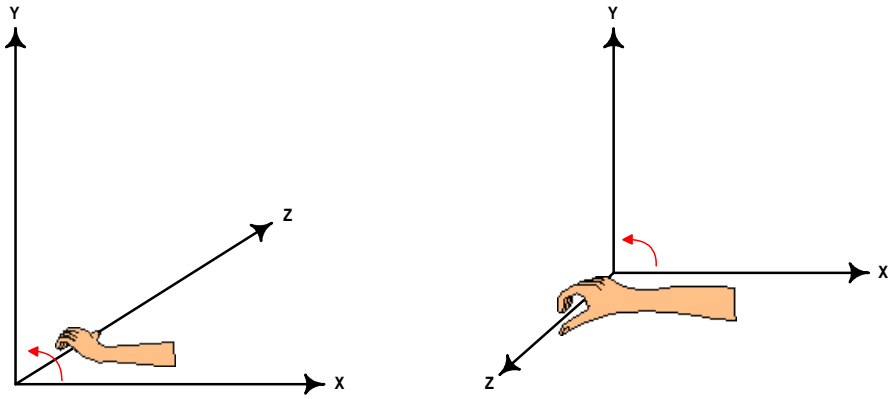
Programming Microsoft® Direct3D® applications requires a working familiarity with 3-D geometric principles. This section introduces the most important geometric concepts for creating 3-D scenes. The following topics are covered.

- 3-D Coordinate Systems
- 3-D Primitives
- Triangle Rasterization Rules

These topics provide you with a high-level understanding of the basic concepts employed by a Direct3D application. For more information about these topics, see Further Information.

3-D Coordinate Systems

Typically 3-D graphics applications use two types of Cartesian coordinate systems: left-handed and right-handed. In both coordinate systems, the positive x-axis points to the right, and the positive y-axis points up. You can remember which direction the positive z-axis points by pointing the fingers of either your left or right hand in the positive x-direction and curling them into the positive y-direction. The direction your thumb points, either toward or away from you, is the direction that the positive z-axis points for that coordinate system. The following illustration shows these two coordinate systems.



[C++]
Microsoft® Direct3D® uses a left-handed coordinate system. If you are porting an application that is based on a right-handed coordinate system, you must make two changes to the data passed to Direct3D.

- Flip the order of triangle vertices so that the system traverses them clockwise from the front. In other words, if the vertices are v0, v1, v2, pass them to Direct3D as v0, v2, v1.
- Use the view matrix to scale world space by –1 in the z-direction. To do this, flip the sign of the _31, _32, _33, and _34 member of the **D3DMATRIX** structure that you use for your view matrix.

To obtain what amounts to a right-handed world, use the **D3DXMatrixPerspectiveRH** and **D3DXMatrixOrthoRH** functions to define the projection transform. However, be careful to use the corresponding **D3DXMatrixLookAtRH** function, reverse the backface-culling order, and lay out the cube maps accordingly.

[Visual Basic]
Microsoft® Direct3D® uses a left-handed coordinate system. If you are porting an application that is based on a right-handed coordinate system, you must make two changes to the data passed to Direct3D.

- Flip the order of triangle vertices so that the system traverses them clockwise from the front. In other words, if the vertices are v0, v1, v2, pass them to Direct3D as v0, v2, v1.
- Use the view matrix to scale world space by –1 in the z-direction. To do this, flip the sign of the _31, _32, _33, and _34 member of the **D3DMATRIX** type that you use for your view matrix.

To obtain what amounts to a right-handed world, use the **D3DXMatrixPerspectiveRH** and **D3DXMatrixOrthoRH** functions to define the projection transform. However, be careful to use the corresponding **D3DXMatrixLookAtRH** function, reverse the backface-culling order, and lay out the cube maps accordingly.

Although left-handed and right-handed coordinates are the most common systems, there is a variety of other coordinate systems used in 3-D software. For example, it is not unusual for 3-D modeling applications to use a coordinate system in which the y-axis points toward or away from the viewer, and the z-axis points up. In this case, right-handedness is defined as any positive axis (x, y, or z) pointing toward the viewer. Left-handedness is defined as any positive axis (x, y, or z) pointing away from the viewer. If you are porting a left-handed modeling application where the z-axis points up, you must do a rotation on all the vertex data in addition to the previous steps.

The essential operations performed on objects defined in a 3-D coordinate system are translation, rotation, and scaling. You can combine these basic transformations to create a transform matrix. For details, see 3-D Transformations.

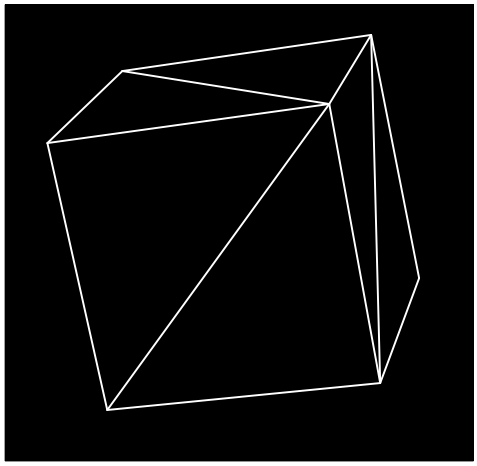
When you combine these operations, the results are not commutative—the order in which you multiply matrices is important.

3-D Primitives

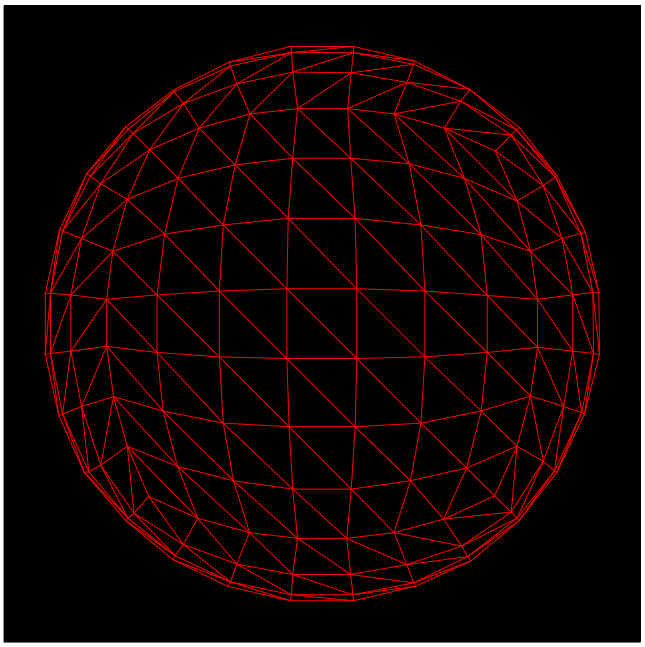
A 3-D primitive is a collection of vertices that form a single 3-D entity. The simplest primitive is a collection of points in a 3-D coordinate system, which is called a point list.

Often, 3-D primitives are polygons. A polygon is a closed 3-D figure delineated by at least three vertices. The simplest polygon is a triangle. Microsoft® Direct3D® uses triangles to compose most of its polygons because all three vertices in a triangle are guaranteed to be coplanar. Rendering nonplanar vertices is inefficient. You can combine triangles to form large, complex polygons and meshes.

The following illustration shows a cube. Two triangles form each face of the cube. The entire set of triangles forms one cubic primitive. You can apply textures and materials to the surfaces of primitives to make them appear to be a single solid form. For details, see Materials and Textures.



You can also use triangles to create primitives whose surfaces appear to be smooth curves. The following illustration shows how a sphere can be simulated with triangles. After a material is applied, the sphere looks curved when it is rendered. This is especially true if you use Gouraud shading. For details, see Gouraud Shading.



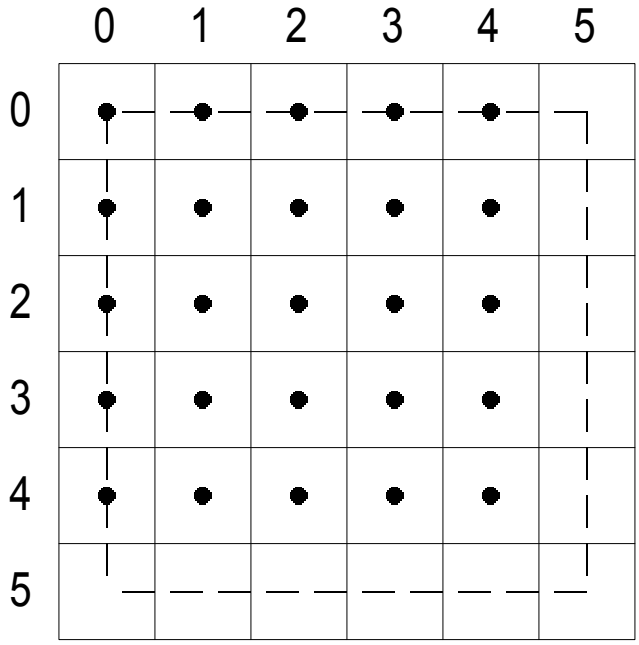
Triangle Rasterization Rules

Often, the points specified for vertices do not precisely match the pixels on the screen. When this happens, Microsoft® Direct3D® applies triangle rasterization rules to decide which pixels apply to a given triangle.

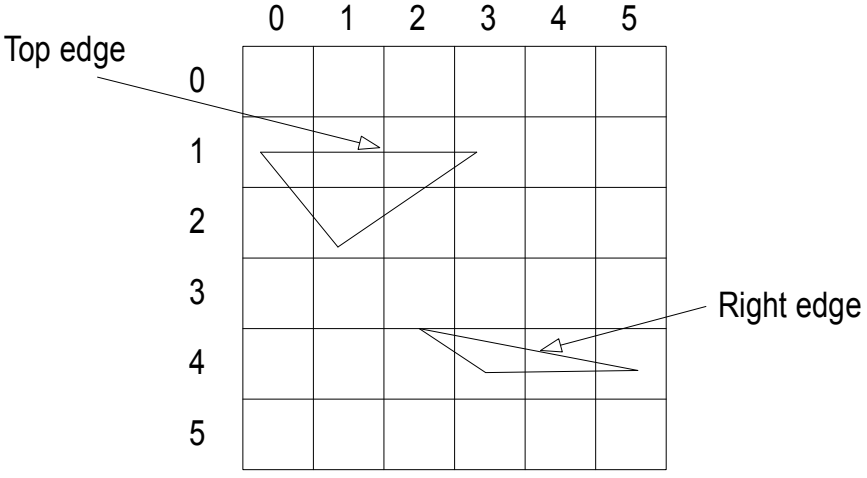
Direct3D uses a top-left filling convention for filling geometry. This is the same convention that is used for rectangles in GDI and OpenGL. In Direct3D, the center of the pixel is the decisive point. If the center is inside a triangle, the pixel is part of the triangle. Pixel centers are at integer coordinates.

This description of triangle-rasterization rules used by Direct3D does not necessarily apply to all available hardware. Your testing may uncover minor variations in the implementation of these rules.

The following illustration shows a rectangle whose upper-left corner is at (0, 0) and whose lower-right corner is at (5, 5). This rectangle fills 25 pixels, just as you would expect. The width of the rectangle is defined as right minus left. The height is defined as bottom minus top.

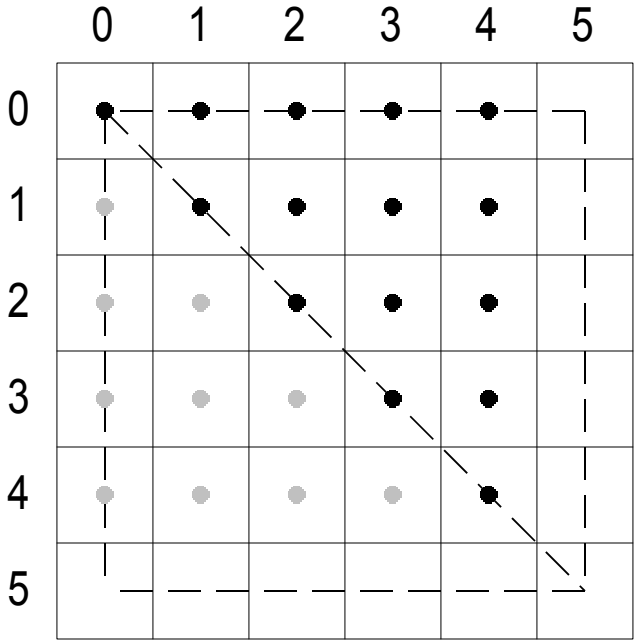


In the top-left filling convention, *top* refers to the vertical location of horizontal spans, and *left* refers to the horizontal location of pixels within a span. An edge cannot be a top edge unless it is horizontal—in general, most triangles have only left and right edges.

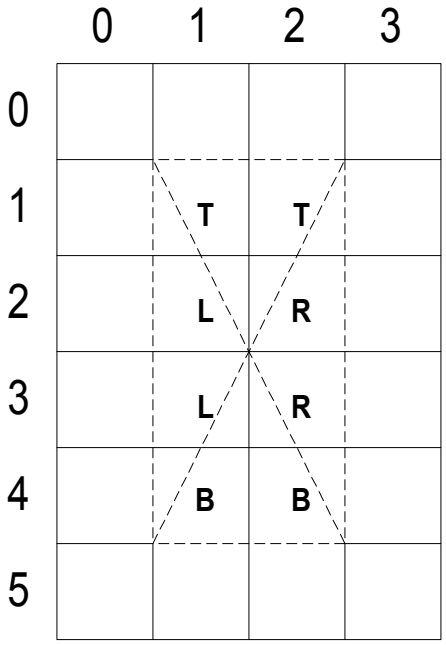


The top-left filling convention determines the action taken by Direct3D when a triangle passes through the center of a pixel. The following illustration shows two triangles, one at (0, 0), (5, 0), and (5, 5), and the other at (0, 5), (0, 0), and (5, 5). The

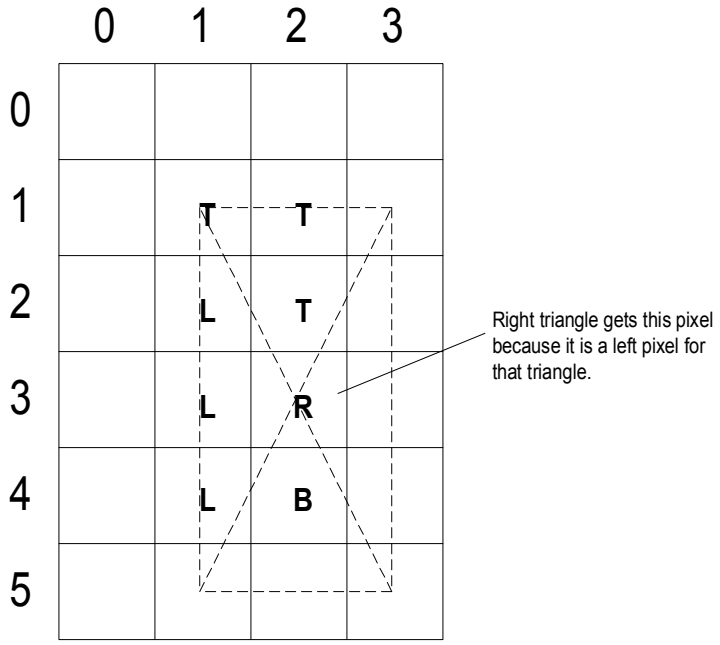
first triangle in this case gets 15 pixels (shown in black), whereas the second gets only 10 pixels (shown in gray) because the shared edge is the left edge of the first triangle.



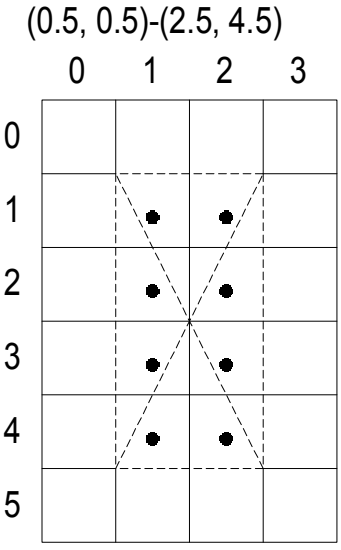
If you define a rectangle with its upper-left corner at (0.5, 0.5) and its lower-right corner at (2.5, 4.5), the center point of this rectangle is at (1.5, 2.5). When the Direct3D rasterizer tessellates this rectangle, the center of each pixel is unambiguously inside each of the four triangles, and the top-left filling convention is not needed. The following illustration shows this. The pixels in the rectangle are labeled according to the triangle in which Direct3D includes them.

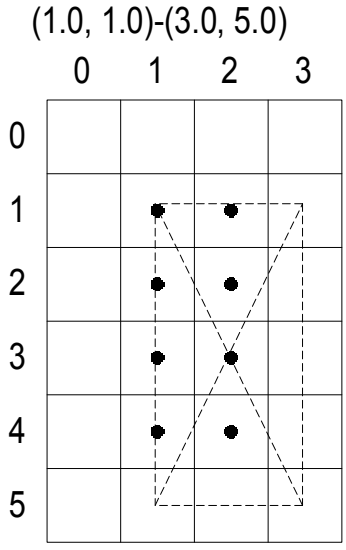


If you move the rectangle in the previous example so that its upper-left corner is at (1.0, 1.0), its lower-right corner at (3.0, 5.0), and its center point at (2.0, 3.0), Direct3D applies the top-left filling convention. Most pixels in this rectangle straddle the border between two or more triangles, as the next illustration shows.



For both rectangles, the same pixels are affected.





Shading Techniques

This section describes techniques used in Microsoft® Direct3D® to control the shading of 3-D polygons.

- Shading Modes
- Comparing Shading Modes
- Setting the Shading Mode
- Face and Vertex Normal Vectors
- Triangle Interpolants

Shading Modes

The shading mode used to render a polygon has a profound effect on its appearance. Shading modes determine the intensity of color and lighting at any point on a polygon face. Microsoft® Direct3D® supports two shading modes:

- Flat Shading
- Gouraud Shading

Flat Shading

In the flat shading mode, the Direct3D rendering pipeline renders a polygon, using the color of the polygon material at its first vertex as the color for the entire polygon. 3-D objects that are rendered with flat shading have visibly sharp edges between polygons if they are not coplanar.

The following figure shows a teapot rendered with flat shading. The outline of each polygon is clearly visible. Flat shading is computationally the least expensive form of shading.

Gouraud Shading

When Microsoft® Direct3D® renders a polygon using Gouraud shading, it computes a color for each vertex by using the vertex normal and lighting parameters. Then, it interpolates the color across the face of the polygons The interpolation is done linearly. For example, if the red component of the color of vertex 1 is 0.8 and the red component of vertex 2 is 0.4, using the Gouraud shading mode and the RGB color model, the Direct3D lighting module assigns a red component of 0.6 to the pixel at the midpoint of the line between these vertices.

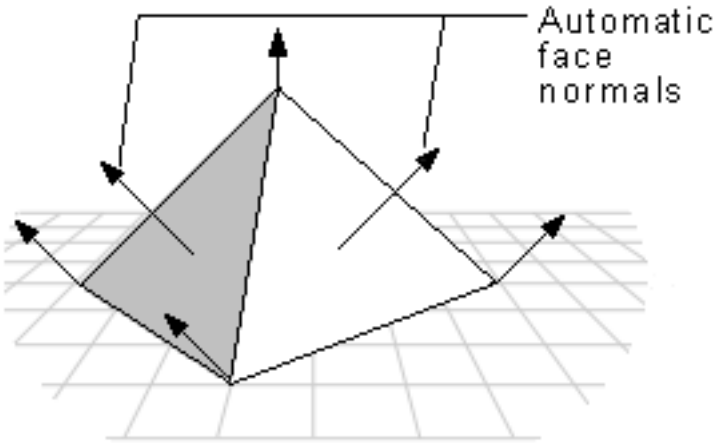
The following figure demonstrates Gouraud shading. This teapot is composed of many flat, triangular polygons. However, Gouraud shading makes the surface of the object appear curved and smooth.

Gouraud shading can also be used to display objects with sharp edges.

For more information, see Face and Vertex Normal Vectors.

Comparing Shading Modes

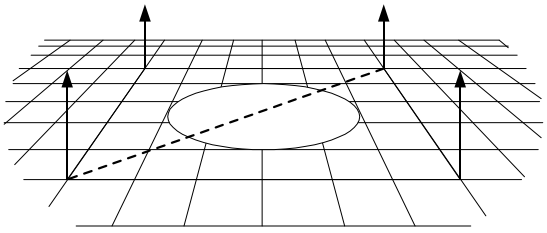
In flat shading mode, the pyramid below is displayed with a sharp edge between adjoining faces. In Gouraud shading mode, however, shading values are interpolated across the edge, and the final appearance is of a curved surface.



Gouraud shading lights flat surfaces more realistically than flat shading. A face in the flat shading mode is a uniform color, but Gouraud shading enables light to fall across a face more correctly. This effect is particularly obvious if there is a nearby point source.

Gouraud shading smoothes the sharp edges between polygons that are visible with flat shading. However, it can result in Mach bands, which are bands of color or light that are not smoothly blended across adjacent polygons. Your application can reduce the appearance of Mach bands by increasing the number of polygons in an object, increasing screen resolution, or increasing the color depth of the application.

Gouraud shading can miss some details. One example is the case shown by the following illustration, in which a spotlight is completely contained within a polygon face.



In this case, Gouraud shading, which interpolates between vertices, would miss the spotlight altogether; the face would be rendered as though the spotlight did not exist.

Setting the Shading Mode

[C++]
Microsoft® Direct3D® enables one shading mode to be selected at a time. By default, Gouraud shading is selected. In C++, you can change the shading mode by calling the **IDirect3DDevice8::SetRenderState** method. Set the *State* parameter to D3DRS_SHADEMODE. The *State* parameter must be set to a member of the **D3DSHADEMODE** enumeration. The following sample code examples illustrate how the current shading mode of a Direct3D application can be set to flat or Gouraud shading mode.

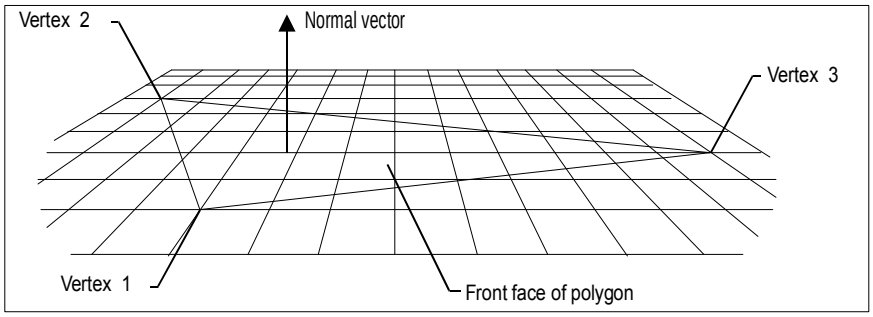
```
// Set to flat shading.
// This code example assumes that pDev is a valid pointer to
// an IDirect3DDevice8 interface.
hr = pDev->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
if(FAILED(hr))
{
   // Code to handle the error goes here.
}

// Set to Gouraud shading. This is the default for Direct3D.
hr = pDev->SetRenderState(D3DRS_SHADEMODE,
                          D3DSHADE_GOURAUD);
if(FAILED(hr))
{
   // Code to handle the error goes here.
}
```

[Visual Basic]
Microsoft® Direct3D® enables one shading mode to be selected at a time. By default, Gouraud shading is selected. In Microsoft Visual Basic®, you can change the shading mode by calling the **Direct3DDevice8.SetRenderState** method. Set the *State* parameter to D3DRS_SHADEMODE. The *State* parameter must be set to a member of the **CONST_D3DSHADEMODE** enumeration. The following code example illustrates how the current shading mode of a Direct3D application can be set to flat or Gouraud shading mode.

```
' Set to flat shading.
' This code example assumes that d3dDev is a valid reference to
' a Direct3DDevice8 object.
On Local Error Resume Next
Call d3dDev.SetRenderState(D3DRS_SHADEMODE, _
                    D3DSHADE_FLAT)

' Check for an error.
If Err.Number <> D3D_OK Then
   ' Handle the error.
End If

' Set to Gouraud shading. this is the default for Direct3D.
Call d3dDev.SetRenderState(D3DRS_SHADEMODE, _
                    D3DSHADE_GOURAUD)

If Err.Number <> D3D_OK Then
   ' Handle the error.
End If
```

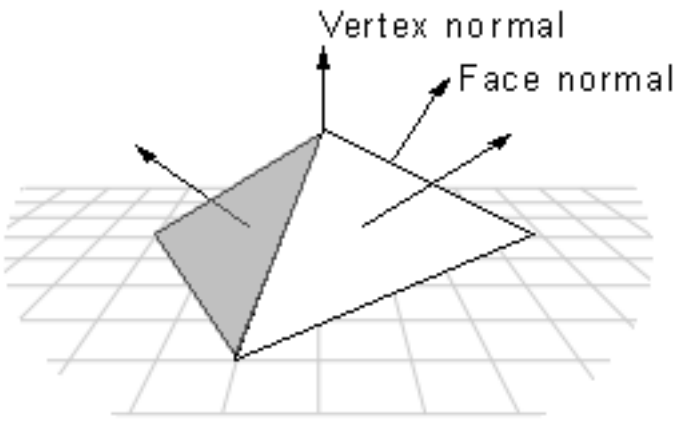
Face and Vertex Normal Vectors

Each face in a mesh has a perpendicular normal vector. The vector's direction is determined by the order in which the vertices are defined and by whether the coordinate system is right- or left-handed. The face normal points away from the front side of the face. In Microsoft® Direct3D®, only the front of a face is visible. A front face is one in which vertices are defined in clockwise order.



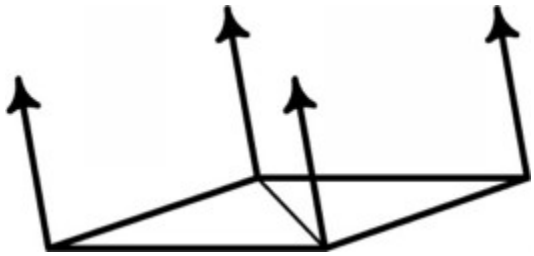
Any face that is not a front face is a back face. Direct3D does not always render back faces; therefore, back faces are said to be culled. You can change the culling mode to render back faces if you want. See Culling State for more information.

Direct3D applications do not need to specify face normals; the system calculates them automatically when they are needed. The system uses face normals in the flat shading mode. In the Gouraud shading mode, Direct3D uses the vertex normal. It also uses the vertex normal for controlling lighting and texturing effects.



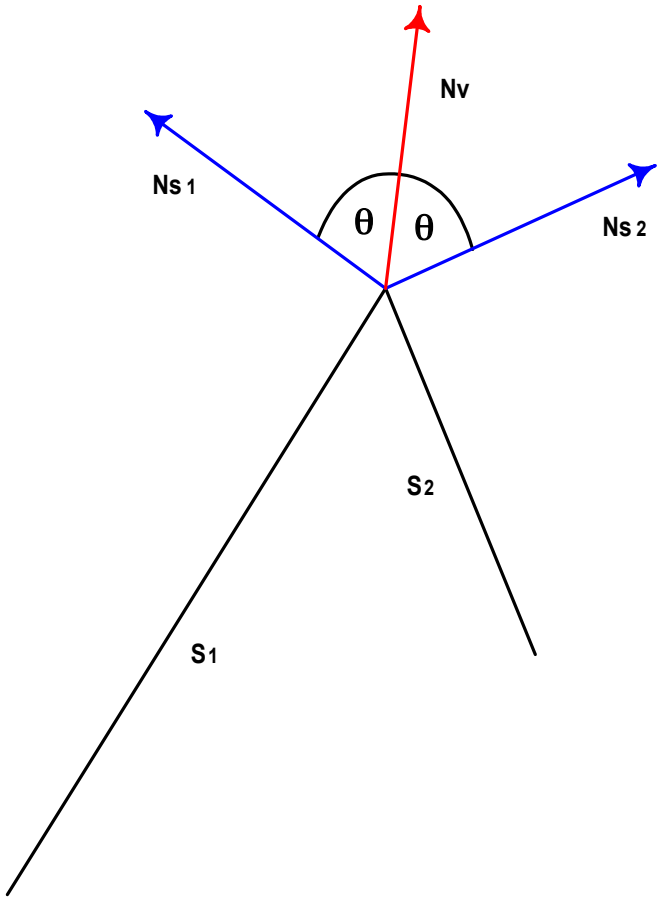
When applying Gouraud shading to a polygon, Direct3D uses the vertex normals to calculate the angle between the light source and the surface. It calculates the color and intensity values for the vertices and interpolates them for every point across all the primitive's surfaces. Direct3D calculates the light intensity value by using the angle. The greater the angle, the less light is shining on the surface.

If you are creating an object that is flat, set the vertex normals to point perpendicular to the surface, as shown in the following illustration. A flat surface composed of two triangles is defined.

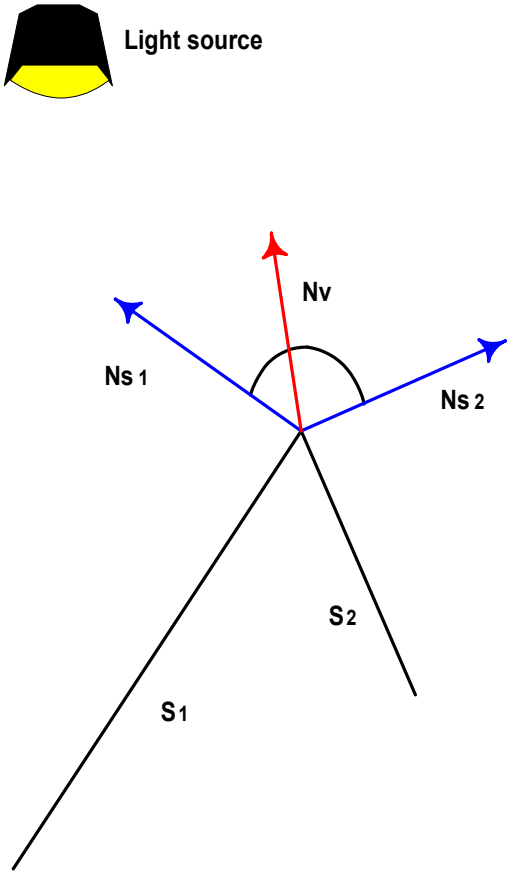


It is more likely, however, that your object is made up of triangle strips and the triangles are not coplanar. One simple way to achieve smooth shading across all the triangles in the strip is to first calculate the surface normal vector for each polygonal face with which the vertex is associated. The vertex normal can be set to make an equal angle with each surface normal. However, this method might not be efficient enough for complex primitives.

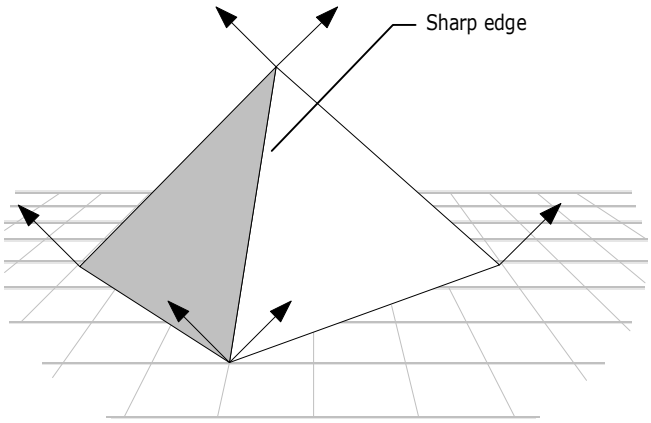
This method is illustrated by the following figure, which shows two surfaces, S1 and S2 seen edge-on from above. The normal vectors for S1 and S2 are shown in blue. The vertex normal vector is shown in red. The angle that the vertex normal vector makes with the surface normal of S1 is the same as the angle between the vertex normal and the surface normal of S2. When these two surfaces are lit and shaded with Gouraud shading, the result is a smoothly shaded, smoothly rounded edge between them.



If the vertex normal leans toward one of the faces with which it is associated, it causes the light intensity to increase or decrease for points on that surface, depending on the angle it makes with the light source. An example is shown in the following figure. Again, these surfaces are seen edge-on. The vertex normal leans toward S1, causing it to have a smaller angle with the light source than if the vertex normal had equal angles with the surface normals.



You can use Gouraud shading to display some objects in a 3-D scene with sharp edges. To do so, duplicate the vertex normal vectors at any intersection of faces where a sharp edge is required, as shown in the following illustration.



If you use the DrawPrimitive methods to render your scene, define the object with sharp edges as a triangle list, rather than a triangle strip. When you define an object as a triangle strip, Direct3D treats it as a single polygon composed of multiple triangular faces. Gouraud shading is applied both across each face of the polygon and between adjacent faces. The result is an object that is smoothly shaded from face to face. Because a triangle list is a polygon composed of a series of disjoint triangular faces, Direct3D applies Gouraud shading across each face of the polygon. However, it is not applied from face to face. If two or more triangles of a triangle list are adjacent, they appear to have a sharp edge between them.

Another alternative is to change to flat shading when rendering objects with sharp edges. This is computationally the most efficient method, but it may result in objects in the scene that are not rendered as realistically as the objects that are Gouraud-shaded.

Triangle Interpolants

The system interpolates the characteristics of a triangle's vertices across the triangle when it renders a face. These are the triangle interpolants:

- Color
- Specular
- Alpha

All the triangle interpolants are modified by the current shading mode, as shown in the following table.

Shading mode	**Description**
Flat	No interpolation is done. Instead, the color of the first vertex in the triangle is applied across the entire face.
Gouraud	Linear interpolation is performed between all three vertices.

The color and specular interpolants are treated differently, depending on the color model. In the RGB color model, the system uses the red, green, and blue color components in the interpolation. In the monochromatic or ramp model, the system uses only the blue component of the vertex color.

The alpha component of a color is treated as a separate interpolant because device drivers can implement transparency in two different ways: by using texture blending or by using stippling.

[C++]
Use the **ShadeCaps** member of the **D3DCAPS8** structure to determine what forms of interpolation the current device driver supports.

[Visual Basic]
Use the **ShadeCaps** member of the **D3DCAPS8** type to determine what forms of interpolation the current device driver supports.

Matrices and Transformations

Use matrices in Microsoft® Direct3D® to define world, view, and projection transformations. If you haven't programmed for 3-D graphics before, this section will help you familiarize yourself with the key concepts you need to understand in order to get started. If you have prior experience in 3-D programming, skip this section, or skim the following topics.

- Matrices
- 3-D Transformations

Matrices

[C++]
Matrices in Microsoft® Direct3D® are represented by a 4´4 homogeneous matrix, defined by the **D3DMATRIX** structure.

[Visual Basic]
Matrices in Microsoft® Direct3D® are represented by a 4×4 homogeneous matrix, defined by the **D3DMATRIX** type.

[C++]
The Direct3DX utility library implementation of the **D3DMATRIX** structure (**D3DXMATRIX**) implements a parentheses ("()") operator. This operator offers

convenient access to values in the matrix for C++ programmers. Instead of having to refer to the structure members by name, you can refer to them by row and column number and index the numbers as needed. These indices are zero-based, so, for example, the element in the third row, second column would be M(2, 1).

You need a basic knowledge of matrices to work with Direct3D. For more information, see 3-D Transformations.

3-D Transformations

Microsoft® Direct3D® uses matrices to perform 3-D transformations. This section explains how matrices create 3-D transformations, describes some common uses for transformations, and details how you can combine matrices to produce a single matrix that encompasses multiple transformations. Information is divided into the following topics.

- About 3-D Transformations
- Translation
- Rotation
- Scaling
- Matrix Concatenation

For more information about transformations in Direct3D, see the Direct3D Rendering Pipeline.

About 3-D Transformations

In applications that work with 3-D graphics, you can use geometrical transformations to do the following:

- Express the location of an object relative to another object.
- Rotate and size objects.
- Change viewing positions, directions, and perspectives.

You can transform any point into another point by using a 4×4 matrix. In the following example, a matrix is reinterprets the point (x, y, z), producing the new point (x', y', z').

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

Perform the following operations on (x, y, z) and the matrix to produce the point (x', y', z').

$$x' = (x \times M_{11}) + (y \times M_{21}) + (z \times M_{31}) + (1 \times M_{41})$$
$$y' = (x \times M_{12}) + (y \times M_{22}) + (z \times M_{32}) + (1 \times M_{42})$$
$$z' = (x \times M_{13}) + (y \times M_{23}) + (z \times M_{33}) + (1 \times M_{43})$$

The most common transformations are translation, rotation, and scaling. You can combine the matrices that produce these effects into a single matrix to calculate several transformations at once. For example, you can build a single matrix to translate and rotate a series of points. For more information, see Matrix Concatenation.

Matrices are written in row-column order. A matrix that evenly scales vertices along each axis, known as uniform scaling, is represented by the following matrix using mathematical notation.

$$\begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[C++]
In C++, Microsoft® Direct3D® declares matrices as a two-dimensional array, using the **D3DMATRIX** structure. The following example shows how to initialize a **D3DMATRIX** structure to act as a uniform scaling matrix.

```
// In this example, s is a variable of type float.

D3DMATRIX scale = {
    s,      0.0f,   0.0f,   0.0f,
    0.0f,   s,      0.0f,   0.0f,
    0.0f,   0.0f,   s,      0.0f,
    0.0f,   0.0f,   0.0f,   1.0f
};
```

[Visual Basic]
In Microsoft® Visual Basic®, Microsoft Direct3D® uses matrices declared as a two-dimensional array, using the **D3DMATRIX** type. The following example shows how to initialize a variable of type **D3DMATRIX** to act as a uniform scaling matrix.

```
' In this example, s is a variable of type Single.

Dim ScaleMatrix As D3DMATRIX

   With ScaleMatrix
      .m11 = s
      .m22 = s
      .m33 = s
      .m44 = 1
   End With
```

Translation

The following transformation translates the point (x, y, z) to a new point (x', y', z').

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

[C++]

You can create a translation matrix by hand in C++. The following example shows the source code for a function that creates a matrix to translate vertices.

```
D3DXMATRIX Translate(const float dx, const float dy, const float dz) {
   D3DXMATRIX ret;

   D3DXMatrixIdentity(&ret);  // Implemented by Direct3DX
   ret(3, 0) = dx;
   ret(3, 1) = dy;
   ret(3, 2) = dz;
   return ret;
}   // End of Translate
```

For convenience, the Direct3DX utility library supplies the **D3DXMatrixTranslation** function.

[Visual Basic]

You can create a translation matrix by hand in Microsoft® Visual Basic®. The following example shows the source code for a function that creates a matrix to translate vertices.

```
Sub TranslateMatrix(m As D3DMATRIX, v As D3DVECTOR)
   D3DXMatrixIdentity m  ' Implemented by Direct3DX.
   m.m41 = v.x
   m.m42 = v.y
   m.m43 = v.z
End Sub
```

For convenience, the Direct3DX utility library supplies the **D3DXMatrixTranslation** function.

Rotation

The transformations described here are for left-handed coordinate systems, and so may be different from transformation matrices that you have seen elsewhere. For more information, see 3-D Coordinate Systems.

The following transformation rotates the point (x, y, z) around the x-axis, producing a new point (x', y', z').

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following transformation rotates the point around the y-axis.

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following transformation rotates the point around the z-axis.

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In these example matrices, the Greek letter theta (θ) stands for the angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

[C++]
In a C++ application, use the **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, and **D3DXMatrixRotationZ** functions supplied by the Direct3DX utility library to create rotation matrices. The following is the code for the **D3DXMatrixRotationX** function.

```
D3DXMATRIX* WINAPI D3DXMatrixRotationX
    ( D3DXMATRIX *pOut, float angle )
{
#if DBG
    if(!pOut)
        return NULL;
#endif

    float sin, cos;
    sincosf(angle, &sin, &cos);  // Determine sin and cos of angle.

    pOut->_11 = 1.0f; pOut->_12 =  0.0f;   pOut->_13 = 0.0f; pOut->_14 = 0.0f;
    pOut->_21 = 0.0f; pOut->_22 =  cos;    pOut->_23 = sin;  pOut->_24 = 0.0f;
    pOut->_31 = 0.0f; pOut->_32 = -sin;    pOut->_33 = cos;  pOut->_34 = 0.0f;
    pOut->_41 = 0.0f; pOut->_42 =  0.0f;   pOut->_43 = 0.0f; pOut->_44 = 1.0f;

    return pOut;
}
```

[Visual Basic]
In a Microsoft® Visual Basic® application, use the **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, and **D3DXMatrixRotationZ** functions supplied by the Direct3DX utility library to create rotation matrices.

If you manually create a matrix for rotation about an axis—in this case, the x-axis—the source code looks something like the following:

```
Sub CreateXRotation(ret As D3DMATRIX, rads As Single)
    Dim cosine As Single
    Dim sine As Single
```

```
    cosine = Cos(rads)
    sine = Sin(rads)

    D3DXMatrixIdentity ret  ' Implemented by Direct3DX.

    ret.m22 = cosine
    ret.m23 = sine
    ret.m32 = -sine
    ret.m33 = cosine
End Sub
```

Scaling

The following transformation scales the point (x, y, z) by arbitrary values in the x-, y-, and z-directions to a new point (x', y', z').

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Concatenation

One advantage of using matrices is that you can combine the effects of two or more matrices by multiplying them. This means that, to rotate a model and then translate it to some location, you don't need to apply two matrices. Instead, you multiply the rotation and translation matrices to produce a composite matrix that contains all their effects. This process, called *matrix concatenation*, can be written with the following formula.

$$C = M_1 \cdot M_2 \cdot M_{n-1} \cdot M_n$$

In this formula, C is the composite matrix being created, and M_1 through M_n are the individual transformations that matrix C contains. In most cases, only two or three matrices are concatenated, but there is no limit.

[C++]
Use the **D3DXMatrixMultiply** function to perform matrix multiplication.

[Visual Basic]
Use the **D3DXMatrixMultiply** function to perform matrix multiplication.

The order in which the matrix multiplication is performed is crucial. The preceding formula reflects the left-to-right rule of matrix concatenation. That is, the visible effects of the matrices that you use to create a composite matrix occur in left-to-right order. A typical world transformation matrix is shown the following example. Imagine that you are creating the world transformation matrix for a stereotypical flying saucer. You would probably want to spin the flying saucer around its center—the y-axis of model space—and translate it to some other location in your scene. To accomplish this effect, you first create a rotation matrix, and then multiply it by a translation matrix, as shown in the following formula.

$$W = R_y \cdot T_w$$

In this formula, R_y is a matrix for rotation about the y-axis, and T_w is a translation to some position in world coordinates.

The order in which you multiply the matrices is important because, unlike multiplying two scalar values, matrix multiplication is not commutative. Multiplying the matrices in the opposite order has the visual effect of translating the flying saucer to its world space position, and then rotating it around the world origin.

No matter what type of matrix you are creating, remember the left-to-right rule to ensure that you achieve the expected effects.

Vectors, Vertices, and Quaternions

Throughout Microsoft® Direct3D®, vertices describe position and orientation. Each vertex in a primitive is described by a vector that gives its position, color, texture coordinates, and a normal vector that gives its orientation.

Quaternions add a fourth element to the [*x*, *y*, *z*] values that define a three-component-vector. Quaternions are an alternative to the matrix methods that are typically used for 3-D rotations. A quaternion represents an axis in 3-D space and a rotation around that axis. For example, a quaternion might represent a (1,1,2) axis and a rotation of 1 radian. Quaternions carry valuable information, but their true power comes from the two operations that you can perform on them: composition and interpolation.

Performing composition on quaternions is similar to combining them. The composition of two quaternions is notated as follows:

$$Q = q_1 \circ q_2$$

The composition of two quaternions applied to a geometry means "rotate the geometry around $axis_2$ by $rotation_2$, then rotate it around $axis_1$ by $rotation_1$." In this case, Q represents a rotation around a single axis that is the result of applying q_2, then q_1 to the geometry.

Using quaternion interpolation, an application can calculate a smooth and reasonable path from one axis and orientation to another. Therefore, interpolation between q_1 and q_2 provides a simple way to animate from one orientation to another.

When you use composition and interpolation together, they provide you with a simple way to manipulate a geometry in a manner that appears complex. For example, imagine that you have a geometry that you want to rotate to a given orientation. You know that you want to rotate it r_2 degrees around $axis_2$, then rotate it r_1 degrees around $axis_1$, but you don't know the final quaternion. By using composition, you could combine the two rotations on the geometry to get a single quaternion that is the result. Then, you could interpolate from the original to the composed quaternion to achieve a smooth transition from one to the other.

[C++]
The Direct3DX utility library includes functions that help you work with quaternions. For example, the **D3DXQuaternionRotationAxis** function adds a rotation value to a vector that defines an axis of rotation, and returns the result in a quaternion defined by a **D3DXQUATERNION** structure. Additionally, the **D3DXQuaternionMultiply** function composes quaternions and the **D3DXQuaternionSlerp** performs spherical linear interpolation between two quaternions.

Direct3D applications can use the following functions to simplify the task of working with quaternions.

- **D3DXQuaternionBaryCentric**
- **D3DXQuaternionConjugate**
- **D3DXQuaternionDot**
- **D3DXQuaternionExp**
- **D3DXQuaternionIdentity**
- **D3DXQuaternionInverse**
- **D3DXQuaternionIsIdentity**
- **D3DXQuaternionLength**
- **D3DXQuaternionLengthSq**
- **D3DXQuaternionLn**
- **D3DXQuaternionMultiply**
- **D3DXQuaternionNormalize**
- **D3DXQuaternionRotationAxis**
- **D3DXQuaternionRotationMatrix**
- **D3DXQuaternionRotationYawPitchRoll**
- **D3DXQuaternionSlerp**

- **D3DXQuaternionSquad**
- **D3DXQuaternionToAxisAngle**

Direct3D applications can use the following functions to simplify the task of working with three-component-vectors.

- **D3DXVec3Add**
- **D3DXVec3BaryCentric**
- **D3DXVec3CatmullRom**
- **D3DXVec3Cross**
- **D3DXVec3Dot**
- **D3DXVec3Hermite**
- **D3DXVec3Length**
- **D3DXVec3LengthSq**
- **D3DXVec3Lerp**
- **D3DXVec3Maximize**
- **D3DXVec3Minimize**
- **D3DXVec3Normalize**
- **D3DXVec3Project**
- **D3DXVec3Scale**
- **D3DXVec3Subtract**
- **D3DXVec3Transform**
- **D3DXVec3TransformCoord**
- **D3DXVec3TransformNormal**
- **D3DXVec3Unproject**

Many additional functions that simplify tasks using two- and four-component-vectors are included among the math functions supplied by the Direct3DX utility library.

[Visual Basic]
The Direct3DX utility library includes functions that help you work with quaternions. For example, the **D3DXQuaternionRotationAxis** function adds a rotation value to a vector that defines an axis of rotation, and returns the result in a quaternion defined by a **D3DQUATERNION** type. Additionally, the **D3DXQuaternionMultiply** function composes quaternions and the **D3DXQuaternionSlerp** performs spherical linear interpolation between two quaternions.

Direct3D applications can use the following functions to simplify the task of working with quaternions.

- **D3DXQuaternionBaryCentric**
- **D3DXQuaternionConjugate**

- **D3DXQuaternionExp**
- **D3DXQuaternionIdentity**
- **D3DXQuaternionInverse**
- **D3DXQuaternionIsIdentity**
- **D3DXQuaternionLength**
- **D3DXQuaternionLengthSq**
- **D3DXQuaternionLn**
- **D3DXQuaternionMultiply**
- **D3DXQuaternionNormalize**
- **D3DXQuaternionRotationAxis**
- **D3DXQuaternionRotationMatrix**
- **D3DXQuaternionRotationYawPitchRoll**
- **D3DXQuaternionSlerp**
- **D3DXQuaternionSquad**
- **D3DXQuaternionToAxisAngle**

Direct3D applications can use the following functions to simplify the task of working with three-component-vectors.

- **D3DXVec3Add**
- **D3DXVec3BaryCentric**
- **D3DXVec3CatmullRom**
- **D3DXVec3Cross**
- **D3DXVec3Dot**
- **D3DXVec3Hermite**
- **D3DXVec3Length**
- **D3DXVec3LengthSq**
- **D3DXVec3Lerp**
- **D3DXVec3Maximize**
- **D3DXVec3Minimize**
- **D3DXVec3Normalize**
- **D3DXVec3Project**
- **D3DXVec3Scale**
- **D3DXVec3Subtract**
- **D3DXVec3Transform**
- **D3DXVec3TransformCoord**
- **D3DXVec3TransformNormal**
- **D3DXVec3Unproject**

Many additional functions that simplify tasks using two- and four-component-vectors are included with the math functions supplied by the Direct3DX utility library.

Copying Surfaces

[C++]
The term blit is shorthand for "bit block transfer," which is the process of transferring blocks of data from one place in memory to another. The blitting device driver interface (DDI) continues to be used in Microsoft® DirectX® 8.0 as the primary mechanism for moving large rectangles of pixels on a per-frame basis, the mechanism behind the copy-oriented **IDirect3DDevice8::Present** method. The transportation of artwork in the blit operation is performed by the **IDirect3DDevice8::UpdateTexture** method. Artwork can also be copied in DirectX 8.0 by using the **IDirect3DDevice8::CopyRects** method, which copies a rectangular subset of pixels.

Note
DirectX 8.0 provides D3DX functions that enable you to load artwork from files, apply color conversion, and resize artwork. For more information on the available functions see Texturing Functions.

[Visual Basic]
The term blit is shorthand for "bit block transfer," which is the process of transferring blocks of data from one place in memory to another. The blitting device driver interface (DDI) continues to be used in Microsoft® DirectX® 8.0 as the primary mechanism for moving large rectangles of pixels on a per-frame basis, the mechanism behind the copy-oriented **Direct3DDevice8.Present** method. The transportation of artwork in the blit operation is performed by the **Direct3DDevice8.UpdateTexture** method. Artwork can also be copied in DirectX 8.0 by using the **Direct3DDevice8.CopyRects** method, which copies a rectangular subset of pixels.

Note
DirectX 8.0 provides D3DX functions that enable you to load artwork from files, apply color conversion, and resize artwork. For more information on the available functions see the D3DX8.class and look under methods for textures.

Page Flipping and Back Buffering

Page flipping is key in multimedia, animation, and game software. Software page flipping is analogous to the way you can do animation with a pad of paper. On each page the artist changes the figure slightly, so that when you flip rapidly between sheets, the drawing appears animated.

Page flipping in software is very similar to this process. Microsoft® Direct3D® implements page flipping functionality through a swap chain which is a property of the device. Initially, you set up a series of Direct3D buffers that are designed to flip to the screen the way the artist's paper flips to the next page. The first buffer is referred to as the color front buffer and the buffers behind it are called back buffers. Your application writes to a back buffer, and then flips the color front buffer so that the back buffer appears on screen. While the system displays the image, your software is again writing to a back buffer. The process continues as long as you are animating, enabling you to animate images quickly and efficiently.

Direct3D makes it easy to set up page flipping schemes, from a relatively simple double-buffered scheme—a color front buffer with one back buffer—to more sophisticated schemes that add additional back buffers.

Rectangles

Throughout Microsoft® Direct3D® and Microsoft Windows® programming, objects on the screen are referred to in terms of bounding rectangles. The sides of a bounding rectangle are always parallel to the sides of the screen, so the rectangle can be described by two points, the top-left corner and bottom-right corner. Most applications use the **RECT** structure to carry information about a bounding rectangle to use when blitting to the screen or performing hit detection.

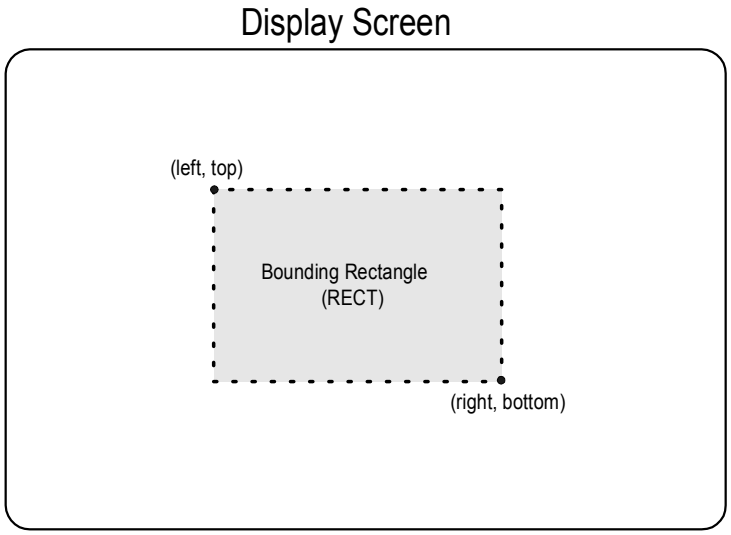
[C++]
In C++, the **RECT** structure has the following definition.

```
typedef struct tagRECT {
    LONG    left;    // This is the top-left corner x-coordinate.
    LONG    top;     // The top-left corner y-coordinate.
    LONG    right;   // The bottom-right corner x-coordinate.
    LONG    bottom;  // The bottom-right corner y-coordinate.
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```

[Visual Basic]
In Microsoft Visual Basic®, the **RECT** type has the following definition.

```
Type RECT
    Left As Long     ' This is the top-left corner x-coordinate.
    Top As Long      ' The top-left corner y-coordinate.
    Right As Long    ' The bottom-right corner x-coordinate.
    Bottom As Long   ' The bottom-right corner y-coordinate.
End Type
```

In the preceding example, the left and top members are the x- and y-coordinates of a bounding rectangle's top-left corner. Similarly, the right and bottom members make up the coordinates of the bottom-right corner. The following diagram illustrates how you can visualize these values.



In the interest of efficiency, consistency, and ease of use, all Direct3D presentation functions work with rectangles.

Understanding DirectX Graphics

This section describes the underlying workings of Microsoft® DirectX® Graphics.

- Direct3D Architecture
- Direct3D Rendering Pipeline

Direct3D Architecture

This section contains general information about the relationship between the Microsoft® Direct3D® component and the rest of Microsoft DirectX®, the operating system, and the system hardware. The following topics are discussed.

- Architectural Overview for Direct3D
- Hardware Abstraction Layer
- System Integration

Architectural Overview for Direct3D

There are two programmable sections of the Microsoft® Direct3D® architecture: vertex shaders and pixel shaders. Vertex shaders are invoked prior to vertex assembly and operate on vectors. Pixel shaders are invoked after any DrawPrimitive calls and operate on pixels.

The following simplified diagram illustrates the Direct3D architecture. It is a subset of the architecture, as Direct3D supports eight sets of textures and texture coordinates.

As shown in this diagram, vertex buffers stream vertex data into the vertex shader. The vertex shader performs geometry operations using the instructions defined by the Direct3DX vertex shader assembler.

The data streamed to the vertex shader does not have to be contained in vertex buffers, but this is the ideal case. After vertex assembly, the assembled vertex data is drawn using DrawPrimitive methods. At this point, each pixel of the drawn primitive is routed to the pixel shader, including its position, color, texture, and texture coordinate. The pixel shader performs pixel operations using the instructions defined by the Direct3DX pixel shader assembler.

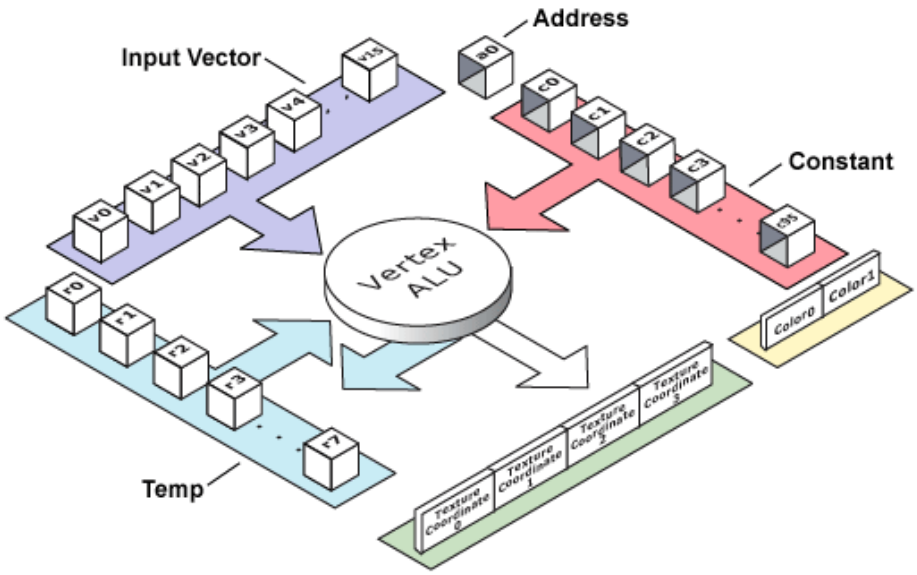
The inputs to the pixel processing pixel shader stage include texture surfaces and the associated texture coordinates controlling the sampling of surfaces. The output pixel is routed to the frame buffer.

For more information on the architecture of the programmable sections of the Direct3D, see Vertex Shader Architecture and Pixel Shader Architecture.

For more information on how vertex and pixel shaders fit into the rendering pipeline, see the Direct3D Rendering Pipeline.

Vertex Shader Architecture

The following diagram illustrates the vertex shader architecture.



The incoming data to the vertex data is taken from vertex data streams, and the constant data is loaded by the vertex shader declarator. For details, see the Vertex Shader Assembler Reference.

Pixel Shader Architecture

The following diagram illustrates the pixel shader architecture.

The incoming data to the pixel shader is the clip space vertex (homogeneous coordinates). The diffuse and specular color comes from the values in the color registers (v0 and v1). The texture coordinates and set textures come from the values in the texture registers (t0, t1, t2, and t3) from the texture setup stage and vertex data. For details, see the Pixel Shader Assembler Reference.

Hardware Abstraction Layer

Microsoft® Direct3D® provides device independence through the hardware abstraction layer (HAL). The HAL is a device-specific interface, provided by the device manufacturer, that Direct3D uses to work directly with the display hardware. Applications never interact with the HAL. Rather, with the infrastructure that the HAL provides, Direct3D exposes a consistent set of interfaces and methods that an application uses to display graphics. The device manufacturer implements the HAL in a combination of 16-bit and 32-bit code under Microsoft Windows®. Under Windows NT® and Windows 2000, the HAL is always implemented in 32-bit code. The HAL can be part of the display driver or a separate dynamic-link library (DLL) that

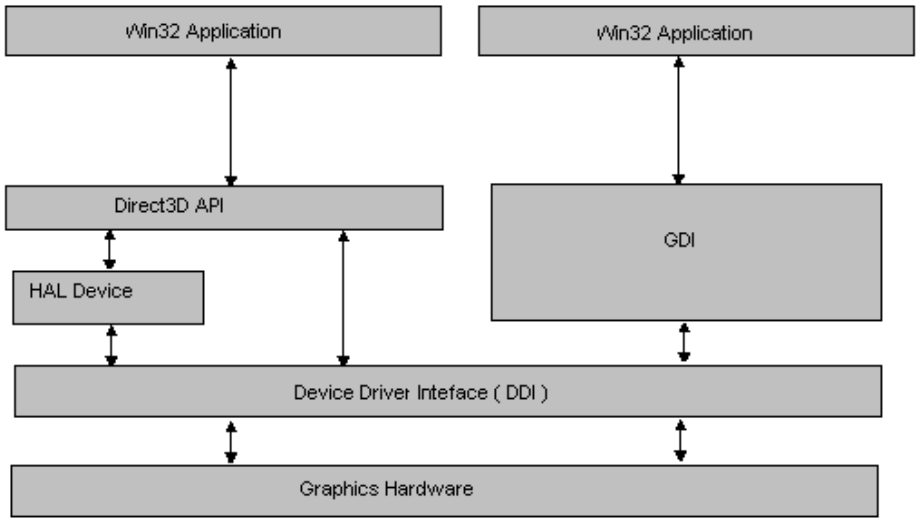
communicates with the display driver through a private interface that driver's creator defines.

The Direct3D HAL is implemented by the chip manufacturer, board producer, or original equipment manufacturer (OEM). The HAL implements only device-dependent code and performs no emulation. If a function is not performed by the hardware, the HAL does not report it as a hardware capability. Additionally, the HAL does not validate parameters; Direct3D does this before the HAL is invoked.

In DirectX 8.0, the HAL can have three different vertex processing modes: software vertex processing, hardware vertex processing, and mixed vertex processing on the same device. The pure device mode is a variant of the HAL device. The pure device type supports hardware vertex processing only, and allows only a small subset of the device state to be queried by the application. Additionally, the pure device is available only on adapters that have a minimum level of capabilities.

System Integration

The following diagram shows the relationships between Microsoft® Direct3D®, the graphics device interface (GDI), the hardware abstraction layer (HAL), and the hardware.



As the preceding diagram shows, Direct3D applications exist alongside GDI applications and both have access to the graphics hardware through the device driver for the graphics card. Unlike GDI, Direct3D can take advantage of hardware features when a HAL device is selected. HAL devices provide hardware acceleration based on the feature set supported by the graphics card. You are provided with a Direct3D method to determine at run time if a device is capable of the task.

The software device, although not supporting the entire DirectX 8.0 feature set, is intended to ensure that applications can always find a working device on any system. However, in many cases an application running on a software device must be prepared to run with a downgraded set of functionality. The capabilities of the software device can be determined in the same fashion as the capabilities of a hardware device.

For more information on devices supported by Direct3D, see Device Types.

Direct3D Rendering Pipeline

The Microsoft® Direct3D® rendering pipeline is a series of processing stages that specify the behavior of the vertex transform and lighting pipeline, and the pixel and texture blending pipeline.

When you design a 3-D application, you can define the world in any units you find convenient, from microns to parsecs. Your application passes a description of that world to Direct3D. This description includes the sizes and relative positions of all the objects in your world and the position and orientation of the viewer. Direct3D transforms these descriptions (vertices) into a series of pixels on the screen. The first step of this process—the transformation of the geometry that you supply into a two-dimensional image—is the geometry pipeline, sometimes called the transformation pipeline. After vertex processing, pixel data is used to perform multitexturing and fog blending. Then alpha, stencil, and depth tests are performed. Finally, frame-buffer blending is done by the rasterizer.

The following topics provide a Direct3D-centered approach to the complete rendering pipeline. The topics introduce key concepts and make parallels from those concepts to their counterparts in the Direct3D API. The fixed-function processing section discusses techniques familiar to developers who have used earlier versions of Microsoft DirectX®, whereas the section on programmable (procedural) processing explains the new procedural model used for vertex and pixel processing in Microsoft DirectX 8.0.

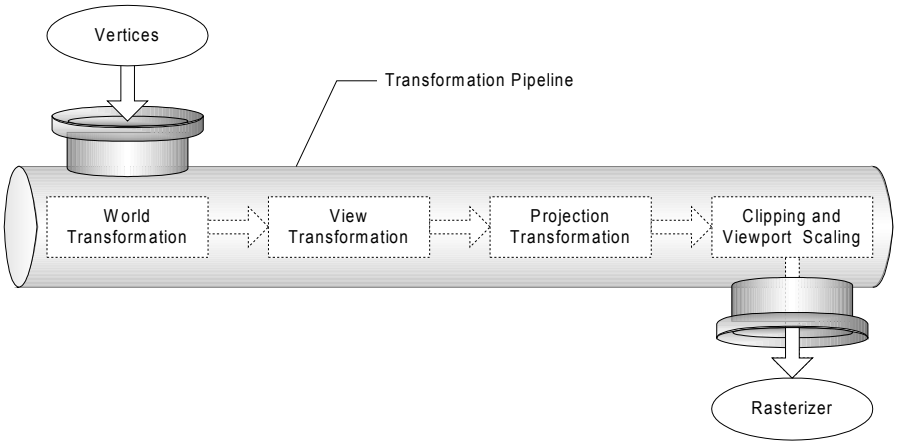
- Fixed Function Vertex and Pixel Processing
- Programmable Vertex and Pixel Processing
- Presenting Vertex Data to the Vertex Processor

Fixed Function Vertex and Pixel Processing

The part of Microsoft® Direct3D® that pushes geometry through the fixed function geometry pipeline is the transformation engine. It locates the model and viewer in the world, projects vertices for display on the screen, and clips vertices to the viewport. The transformation engine also performs lighting computations to determine diffuse and specular components at each vertex.

The geometry pipeline takes vertices as input. The transformation engine applies three transformations—the world, view, and projection transformations—to the

vertices, clips the result, and passes everything to the rasterizer. The following image illustrates the sequence of steps.



At the head of the pipeline, no transformations have been applied, so all of a model's vertices are declared relative to a local coordinate system—this is a local origin and an orientation. This orientation of coordinates is often referred to as model space, and individual coordinates are called model coordinates.

The first stage of the geometry pipeline transforms a model's vertices from their local coordinate system to a coordinate system that is used by all the objects in a scene. The process of reorienting the vertices is called the world transformation. This new orientation is commonly referred to as world space, and each vertex in world space is declared using world coordinates.

In the next stage, the vertices that describe your 3-D world are oriented with respect to a camera. That is, your application chooses a point-of-view for the scene, and world space coordinates are relocated and rotated around the camera's view, turning world space into camera space. This is the view transformation.

The next stage is the projection transformation. In this part of the pipeline, objects are usually scaled with relation to their distance from the viewer in order to give the illusion of depth to a scene; close objects are made to appear larger than distant objects, and so on. For simplicity, this documentation refers to the space in which vertices exist after the projection transformation as projection space. Some graphics books might refer to projection space as post-perspective homogeneous space. Not all projection transformations scale the size of objects in a scene. A projection such as this is sometimes called an affine or orthogonal projection.

In the final part of the pipeline, any vertices that will not be visible on the screen are removed, so that the rasterizer doesn’t take the time to calculate the colors and shading for something that will never be seen. This process is called clipping. After clipping, the remaining vertices are scaled according to the viewport parameters and converted into screen coordinates. The resulting vertices—seen on the screen when the scene is rasterized—exist in screen space.

Information on the fixed vertex and pixel processing pipeline is organized into the following topics.

- Transformation and Lighting Engine
- Viewports and Clipping
- Rasterization

Transformation and Lighting Engine

The following topics explain the mechanics behind each component of the fixed function transformation and lighting engine in detail.

- The World Transformation
- The View Transformation
- The Projection Transformation
- The Lighting Engine
- Mathematics of Direct3D Lighting

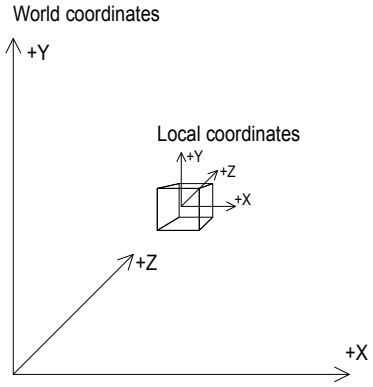
The World Transformation

The discussion of the world transformation introduces basic concepts and provides details on how to set up a world transformation matrix in a Microsoft® Direct3D® application. This information is organized into the following topics.

- What Is the World Transformation?
- Setting Up a World Matrix

What Is the World Transformation?

The world transformation changes coordinates from model space, where vertices are defined relative to a model's local origin, to world space, where vertices are defined relative to an origin common to all the objects in a scene. In essence, the world transformation places a model into the world; hence its name. The following diagram illustrates the relationship between the world coordinate system and a model's local coordinate system.



The world transformation can include any combination of translations, rotations, and scalings. For a discussion of the mathematics of transformations, see 3-D Transformations.

Setting Up a World Matrix

As with any other transformation, you create the world transformation by concatenating a series of transformation matrices into a single matrix that contains the sum total of their effects. In the simplest case, when a model is at the world origin and its local coordinate axes are oriented the same as world space, the world matrix is the identity matrix. More commonly, the world matrix is a combination of a translation into world space and possibly one or more rotations to turn the model as needed.

[C++]

The following example, from a fictitious 3-D model class written in C++, uses the helper functions included in the Direct3DX utility library to create a world matrix that includes three rotations to orient a model and a translation to relocate it relative to its position in world space.

```
/*
 * For the purposes of this example, the following variables
 * are assumed to be valid and initialized.
 *
 * The m_xPos, m_yPos, m_zPos variables contain the model's
 * location in world coordinates.
 *
 * The m_fPitch, m_fYaw, and m_fRoll variables are floats that
 * contain the model's orientation in terms of pitch, yaw, and roll
 * angles, in radians.
 */

void C3DModel::MakeWorldMatrix( D3DXMATRIX* pMatWorld )
{
    D3DXMATRIX MatTemp;  // Temp matrix for rotations.
    D3DXMATRIX MatRot;   // Final rotation matrix, applied to
```

```
                    // pMatWorld.

    // Using the left-to-right order of matrix concatenation,
    // apply the translation to the object's world position
    // before applying the rotations.
    D3DXMatrixTranslation(pMatWorld, m_xPos, m_yPos, m_zPos);
    D3DXMatrixIdentity(&MatRot);

    // Now, apply the orientation variables to the world matrix
    if(m_fPitch || m_fYaw || m_fRoll) {
        // Produce and combine the rotation matrices.
        D3DXMatrixRotationX(&MatTemp, m_fPitch);        // Pitch
        D3DXMatrixMultiply(&MatRot, &MatRot, &MatTemp);
        D3DXMatrixRotationY(&MatTemp, m_fYaw);          // Yaw
        D3DXMatrixMultiply(&MatRot, &MatRot, &MatTemp);
        D3DXMatrixRotationZ(&MatTemp, m_fRoll);         // Roll
        D3DXMatrixMultiply(&MatRot, &MatRot, &MatTemp);

        // Apply the rotation matrices to complete the world matrix.
        D3DXMatrixMultiply(pMatWorld, &MatRot, pMatWorld);
    }
}
```

After you prepare the world transformation matrix, call the **IDirect3DDevice8::SetTransform** method to set it, specifying the **D3DTS_WORLD** macro for the first parameter. For more information, see Setting Transformations.

[Visual Basic]

The following example, from a fictitious 3-D model class written in Microsoft ® Visual Basic® uses the helper functions included in the Direct3DX utility library to create a world matrix that includes three rotations to orient a model and a translation to relocate it relative to its position in world space.

```
'
' For the purposes of this example, the following variables
' are assumed to be valid and initialized.
'
' The m_xPos, m_yPos, m_zPos variables contain the model's
' location in world coordinates.
'
' The m_sPitch, m_sYaw, and m_sRoll variables are Singles that
' contain the model's orientation in terms of pitch, yaw, and roll
' angles, in radians.
'
```

```
Public Function MakeWorldMatrix() As D3DMATRIX
    Dim matWorld As D3DMATRIX   ' World matrix to return.
    Dim matTemp As D3DMATRIX    ' Temp matrix to hold rotations.
    Dim matRot As D3DMATRIX     ' Temp rotation matrix.

    ' Modify matWorld to create a translation matrix.
    D3DXMatrixIdentity matWorld
    matWorld.m41 = m_xPos
    matWorld.m42 = m_yPos
    matWorld.m43 = m_zPos

    D3DXMatrixIdentity matRot ' Sets up the rotation matrix.

    '
    ' Using the left-to-right order of matrix concatenation,
    ' apply the translation to the object's world position
    ' before applying the rotations.
    '

    ' Produce and combine the rotation matrices.
    If (m_sPitch <> 0) Or (m_sYaw <> 0) Or (m_sRoll <> 0) Then
      ' First, pitch.
       D3DXMatrixRotationX matTemp, m_sPitch
       D3DXMatrixMultiply matRot, matRot, matTemp

      ' Then, yaw.
       D3DXMatrixRotationY matTemp, m_sYaw
       D3DXMatrixMultiply matRot, matRot, matTemp

       ' Finally, roll.
       D3DXMatrixRotationZ matTemp, m_sRoll
       D3DXMatrixMultiply matRot, matRot, matTemp

       ' Apply the rotation matrices to the translation already in
       ' matWorld to complete the world matrix.
       D3DXMatrixMultiply matWorld, matRot, matWorld
    End If

    MakeWorldMatrix = matWorld
End Function
```

After you prepare the world transformation matrix, call the **Direct3DDevice8.SetTransform** method to set it, specifying the D3DTS_WORLD flag in the first parameter. For more information, see Setting Transformations.

Note

Microsoft® Direct3D® uses the world and view matrices that you set to configure several internal data structures. Each time you set a new world or view matrix, the system recalculates the associated internal structures. Setting these matrices frequently—for example, thousands of times per frame—is computationally expensive. You can minimize the number of required calculations by concatenating your world and view matrices into a world-view matrix that you set as the world matrix, and then setting the view matrix to the identity. Keep cached copies of individual world and view matrices so that you can modify, concatenate, and reset the world matrix as needed. For clarity, in this documentation Direct3D samples rarely employ this optimization.

The View Transformation

This section introduces the basic concepts of the view transformation and provides details on how to set up a view transformation matrix in a Microsoft® Direct3D® application. This information is organized into the following topics.

- What Is the View Transformation?
- Setting Up a View Matrix

What Is the View Transformation?

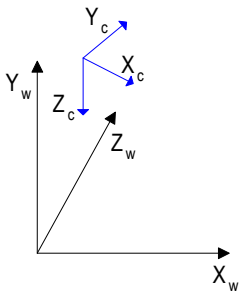
The view transformation locates the viewer in world space, transforming vertices into camera space. In camera space, the camera, or viewer, is at the origin, looking in the positive z-direction. Recall that Microsoft® Direct3D® uses a left-handed coordinate system, so z is positive into a scene. The view matrix relocates the objects in the world around a camera's position—the origin of camera space—and orientation.

There are many ways to create a view matrix. In all cases, the camera has some logical position and orientation in world space that is used as a starting point to create a view matrix that will be applied to the models in a scene. The view matrix translates and rotates objects to place them in camera space, where the camera is at the origin. One way to create a view matrix is to combine a translation matrix with rotation matrices for each axis. In this approach, the following general matrix formula applies.

$$V = T \cdot R_z \cdot R_y \cdot R_x$$

In this formula, V is the view matrix being created, T is a translation matrix that repositions objects in the world, and R_x through R_z are rotation matrices that rotate objects along the x-, y-, and z-axis. The translation and rotation matrices are based on the camera's logical position and orientation in world space. So, if the camera's logical position in the world is <10,20,100>, the aim of the translation matrix is to move objects -10 units along the x-axis, -20 units along the y-axis, and -100 units along the z-axis. The rotation matrices in the formula are based on the camera's orientation, in terms of how the much the axes of camera space are rotated out of alignment with world space. For example, if the camera mentioned earlier is pointing

straight down, its z-axis is 90 degrees (pi/2 radians) out of alignment with the z-axis of world space, as shown in the following illustration.



The rotation matrices apply rotations of equal, but opposite, magnitude to the models in the scene. The view matrix for this camera includes a rotation of -90 degrees around the x-axis. The rotation matrix is combined with the translation matrix to create a view matrix that adjusts the position and orientation of the objects in the scene so that their top is facing the camera, giving the appearance that the camera is above the model.

[C++]
Another approach involves creating the composite view matrix directly. The **D3DXMatrixLookAtLH** and **D3DXMatrixLookAtRH** helper functions use this technique. This approach uses the camera's world space position and a look-at point in the scene to derive vectors that describe the orientation of the camera space coordinate axes. The camera position is subtracted from the look-at point to produce a vector for the camera's direction vector (vector *n*). Then the cross product of the vector *n* and the y-axis of world space is taken and normalized to produce a right vector (vector *u*). Next, the cross product of the vectors *u* and *n* is taken to determine an up vector (vector *v*). The right (*u*), up (*v*), and view-direction (*n*) vectors describe the orientation of the coordinate axes for camera space in terms of world space. The x, y, and z translation factors are computed by taking the negative of the dot product between the camera position and the *u*, *v*, and *n* vectors.

[Visual Basic]
Another approach involves creating the composite view matrix directly. The **D3DXMatrixLookAtRH** and **D3DXMatrixLookAtRH** methods use this technique. This approach uses the camera's world space position and a look-at point in the scene to derive vectors that describe the orientation of the camera space coordinate axes. The camera position is subtracted from the look-at point to produce a vector for the camera's direction vector (vector *n*). Then the cross product of the vector *n* and the y-axis of world space is taken and normalized to produce a right vector (vector *u*). Next, the cross product of the vectors *u* and *n* is taken to determine an up vector (vector *v*). The right (*u*), up (*v*), and view-direction (*n*) vectors describe the orientation of the coordinate axes for camera space in terms of world space. The x, y, and z translation

factors are computed by taking the negative of the dot product between the camera position and the *u*, *v*, and *n* vectors.

These values are put into the following matrix to produce the view matrix.

$$\begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ -(u \cdot c) & -(v \cdot c) & -(n \cdot c) & 1 \end{bmatrix}$$

In this matrix, *u*, *v*, and *n* are the up, right, and view-direction vectors, and *c* is the camera's world space position. This matrix contains all the elements needed to translate and rotate vertices from world space to camera space. After creating this matrix, you can also apply a matrix for rotation around the z-axis to allow the camera to roll.

For information on implementing this technique, see Setting Up a View Matrix.

Setting Up a View Matrix

[C++]
The **D3DXMatrixLookAtLH** and **D3DXMatrixLookAtRH** helper functions create a view matrix based on the camera location and a look-at point. They use the **D3DXVec3Cross**, **D3DXVec3Dot**, **D3DXVec3Normalize**, and **D3DXVec3Subtract** helper functions.

The following code example, illustrates the **D3DXMatrixLookAtLH** function.

```
D3DXMATRIX* WINAPI D3DXMatrixLookAtLH
    ( D3DXMATRIX *pOut, const D3DXVECTOR3 *pEye, const D3DXVECTOR3 *pAt,
      const D3DXVECTOR3 *pUp )
{
#if DBG
    if(!pOut || !pEye || !pAt || !pUp)
        return NULL;
#endif

    D3DXVECTOR3 XAxis, YAxis, ZAxis;

    // Get the z basis vector, which points straight ahead; the
    // difference from the eye point to the look-at point. This is the
    // direction of the gaze (+z).
    D3DXVec3Subtract(&ZAxis, pAt, pEye);

    // Normalize the z basis vector.
```

```
    D3DXVec3Normalize(&ZAxis, &ZAxis);

    // Compute the orthogonal axes from the cross product of the gaze
    // and the pUp vector.
    D3DXVec3Cross(&XAxis, pUp, &ZAxis);
    D3DXVec3Normalize(&XAxis, &XAxis);
    D3DXVec3Cross(&YAxis, &ZAxis, &XAxis);

    // Start building the matrix. The first three rows contain the
    // basis vectors used to rotate the view to point at the look-at
    // point. The fourth row contains the translation values.
    // Rotations are still about the eyepoint.
    pOut->_11 = XAxis.x;
    pOut->_21 = XAxis.y;
    pOut->_31 = XAxis.z;
    pOut->_41 = -D3DXVec3Dot(&XAxis, pEye);

    pOut->_12 = YAxis.x;
    pOut->_22 = YAxis.y;
    pOut->_32 = YAxis.z;
    pOut->_42 = -D3DXVec3Dot(&YAxis, pEye);

    pOut->_13 = ZAxis.x;
    pOut->_23 = ZAxis.y;
    pOut->_33 = ZAxis.z;
    pOut->_43 = -D3DXVec3Dot(&ZAxis, pEye);

    pOut->_14 = 0.0f;
    pOut->_24 = 0.0f;
    pOut->_34 = 0.0f;
    pOut->_44 = 1.0f;

    return pOut;
}
```

As with the world transformation, you call the **IDirect3DDevice8::SetTransform** method to set the view transformation, specifying the D3DTS_VIEW flag in the first parameter. For more information, see Setting Transformations.

[Visual Basic]
Microsoft® Visual Basic® applications use the **D3DXMatrixLookAtLH** and **D3DXMatrixLookAtRH** helper functions to create a view matrix based on the camera location, a look-at point, and an up vector (usually 0,1,0).

The following example code, written in Visual Basic, shows how you can use the **D3DXMatrixLookAtLH** function.

```
' This example assumes that g_D3DDevice is a valid reference
' to a Direct3DDevice8 object.

Dim matView As D3DMATRIX

' Set the eyepoint five units back along the z-axis
' and up three units.
D3DXMatrixLookAtLH matView, vec3(0#, 3#, -5#), _
                    vec3(0#, 0#, 0#), _
                    vec3(0#, 1#, 0#)

g_D3DDevice.SetTransform D3DTS_VIEW, matView
```

As with the world transformation, you call the **Direct3DDevice8.SetTransform** method to set the view transformation, specifying the D3DTS_VIEW flag in the first parameter. See Setting Transformations, for more information.

Performance Optimization Note

Microsoft® Direct3D® uses the world and view matrices that you set to configure several internal data structures. Each time you set a new world or view matrix, the system recalculates the associated internal structures. Setting these matrices frequently—for example, 20,000 times per frame—is computationally expensive. You can minimize the number of required calculations by concatenating your world and view matrices into a world-view matrix that you set as the world matrix, and then setting the view matrix to the identity. Keep cached copies of individual world and view matrices that you can modify, concatenate, and reset the world matrix as needed. For clarity, Direct3D samples rarely employ this optimization.

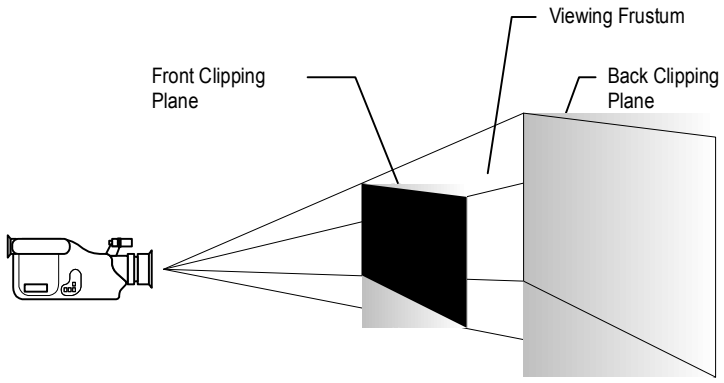
The Projection Transformation

You can think of the projection transformation as controlling the camera's internals; it is analogous to choosing a lens for the camera. This is the most complicated of the three transformation types. This discussion of the projection transformation is organized into the following topics.

- The Viewing Frustum
- What Is the Projection Transformation?
- Setting Up a Projection Matrix
- A W-Friendly Projection Matrix

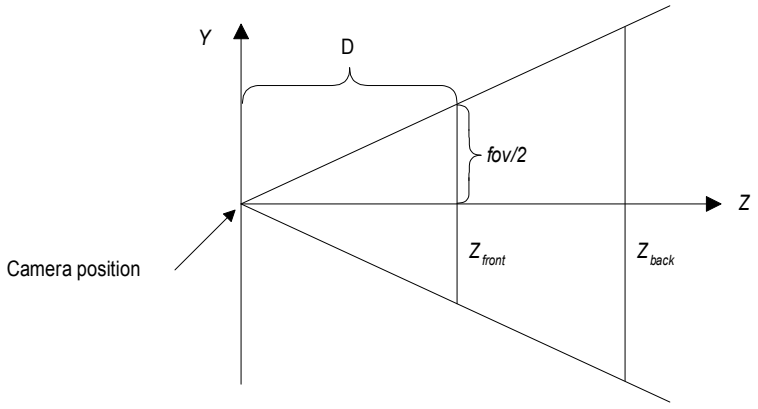
The Viewing Frustum

A viewing frustum is 3-D volume in a scene positioned relative to the viewport's camera. The shape of the volume affects how models are projected from camera space onto the screen. The most common type of projection, a perspective projection, is responsible for making objects near the camera appear bigger than objects in the distance. For perspective viewing, the viewing frustum can be visualized as a pyramid, with the camera positioned at the tip. This pyramid is intersected by a front and back clipping plane. The volume within the pyramid between the front and back clipping planes is the viewing frustum. Objects are visible only when they are in this volume.



If you imagine that you are standing in a dark room and looking through a square window, you are visualizing a viewing frustum. In this analogy, the near clipping plane is the window, and the back clipping plane is whatever finally interrupts your view—the skyscraper across the street, the mountains in the distance, or nothing at all. You can see everything inside the truncated pyramid that starts at the window and ends with whatever interrupts your view, and you can see nothing else.

The viewing frustum is defined by *fov* (field of view) and by the distances of the front and back clipping planes, specified in z-coordinates.



In this illustration, the variable *D* is the distance from the camera to the origin of the space that was defined in the last part of the geometry pipeline—the viewing transformation. This is the space around which you arrange the limits of your viewing frustum. For information about how this *D* variable is used to build the projection matrix, see What Is the Projection Transformation?

What Is the Projection Transformation?

The projection matrix is typically a scale and perspective projection. The projection transformation converts the viewing frustum into a cuboid shape. Because the near end of the viewing frustum is smaller than the far end, this has the effect of expanding objects that are near to the camera; this is how perspective is applied to the scene.

In The Viewing Frustum, the distance between the camera required by the projection transformation and the origin of the space defined by the viewing transformation is defined as *D*. A beginning for a matrix defining the perspective projection might use this *D* variable like this:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

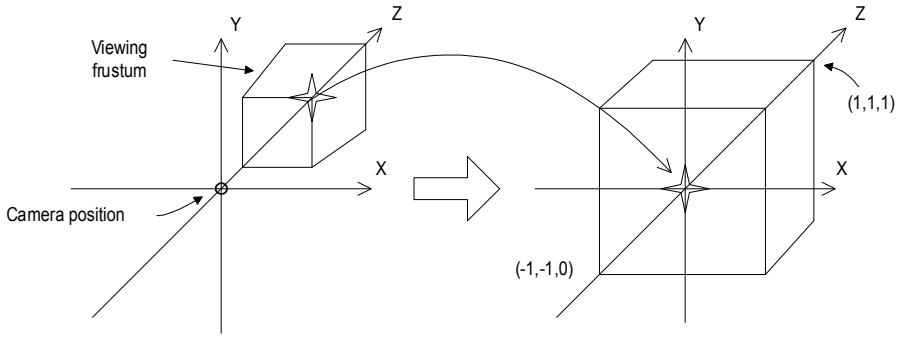
The viewing matrix puts the camera at the origin of the scene. Because the projection matrix needs to have the camera at (0, 0, -*D*), it translates the vector by -*D* in the z-direction, by using the following matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -D & 1 \end{bmatrix}$$

Multiplying these two matrices gives the following composite matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & -D & 0 \end{bmatrix}$$

The following illustration shows how the perspective transformation converts a viewing frustum into a new coordinate space. Notice that the frustum becomes cuboid and also that the origin moves from the upper-right corner of the scene to the center.



In the perspective transformation, the limits of the x- and y-directions are -1 and 1. The limits of the z-direction are 0 for the front plane and 1 for the back plane.

This matrix translates and scales objects based on a specified distance from the camera to the near clipping plane, but it doesn't consider the field of view (*fov*), and the z-values that it produces for objects in the distance can be nearly identical, making depth comparisons difficult. The following matrix addresses these issues, and it adjusts vertices to account for the aspect ratio of the viewport, making it a good choice for the perspective projection.

$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -QZ_n & 0 \end{bmatrix}$$

In this matrix, Z_n is the z-value of the near clipping plane. The variables w, h, and Q have the following meanings. Note that fov_w and fov_h represent the viewport's horizontal and vertical fields of view, in radians.

$$w = \cot\left(fov_w / 2\right)$$

$$h = \cot\left(fov_h / 2\right)$$

$$Q = \frac{Z_f}{Z_f - Z_n}$$

For your application, using field-of-view angles to define the x and y scaling coefficients might not be as convenient as using the viewport's horizontal and vertical dimensions (in camera space). As the math works out, the following two formulas for w and h use the viewport's dimensions, and are equivalent to the preceding formulas.

$$w = \frac{2 \cdot Z_n}{V_w}$$

$$h = \frac{2 \cdot Z_n}{V_h}$$

In these formulas, Z_n represents the position of the near clipping plane, and the V_w and V_h variables represent the width and height of the viewport, in camera space.

[C++]
For a C++ application, these two dimensions correspond directly to the **Width** and **Height** members of the **D3DVIEWPORT8** structure.

Whatever formula you decide to use, it's important that you set Z_n to as large a value as possible, as z-values extremely close to the camera don't vary by much. This makes depth comparisons using 16-bit z-buffers somewhat complicated.

As with the world and view transformations, you call the **IDirect3DDevice8::SetTransform** method to set the projection transformation; for more information, see Setting Transformations.

[Visual Basic]
For a Microsoft® Visual Basic® application, these two dimensions correspond directly to the **Width** and **Height** members of the **D3DVIEWPORT8** type.

Whatever formula you decide to use, it's important that you set Z_n to as large a value as possible, as z-values extremely close to the camera don't vary by much, making depth comparisons using 16-bit z-buffers tricky.

As with the world and view transformations, you call the **Direct3DDevice8.SetTransform** method to set the projection transformation; for more information, see Setting Transformations.

Setting Up a Projection Matrix

[C++]
The following ProjectionMatrix sample function—written in C++—takes four input parameters that set the front and back clipping planes, as well as the horizontal and vertical field of view angles. This code parallels the approach discussed in the What Is the Projection Transformation? topic. The fields of view should be less than pi radians.

```
D3DMATRIX
ProjectionMatrix(const float near_plane, // Distance to near clipping
                                         // plane
                 const float far_plane,  // Distance to far clipping
                                         // plane
                 const float fov_horiz,  // Horizontal field of view
                                         // angle, in radians
                 const float fov_vert)   // Vertical field of view
                                         // angle, in radians
{
    float    h, w, Q;

    w = (float)1/tan(fov_horiz*0.5);  // 1/tan(x) == cot(x)
    h = (float)1/tan(fov_vert*0.5);   // 1/tan(x) == cot(x)
    Q = far_plane/(far_plane - near_plane);

    D3DMATRIX ret;
    ZeroMemory(&ret, sizeof(ret));

    ret(0, 0) = w;
    ret(1, 1) = h;
    ret(2, 2) = Q;
    ret(3, 2) = -Q*near_plane;
    ret(2, 3) = 1;
```

```
    return ret;
}   // End of ProjectionMatrix
```

After you create the matrix, you must set it in a call to the **IDirect3DDevice8::SetTransform** method, specifying D3DTRANSFORMSTATE_PROJECTION in the first parameter. For details, see Setting Transformations.

The Direct3DX utility library provides the following functions to help you set up your projections matrix.

- **D3DXMatrixPerspectiveLH**
- **D3DXMatrixPerspectiveRH**
- **D3DXMatrixPerspectiveFovLH**
- **D3DXMatrixPerspectiveFovRH**
- **D3DXMatrixPerspectiveOffCenterLH**
- **D3DXMatrixPerspectiveOffCenterRH**

[Visual Basic]
Applications written in Microsoft® Visual Basic® can create a projection matrix as described in the What Is the Projection Transformation? topic. However, the Direct3DX utility library provides the following functions to help you set up your projections matrix.

- **D3DXMatrixPerspectiveLH**
- **D3DXMatrixPerspectiveRH**
- **D3DXMatrixPerspectiveFovLH**
- **D3DXMatrixPerspectiveFovRH**
- **D3DXMatrixPerspectiveOffCenterLH**
- **D3DXMatrixPerspectiveOffCenterRH**

The following Visual Basic code example shows how **D3DXMatrixPerspectiveLH** is commonly used.

```
' This example assumes that g_D3DDevice is a valid reference
' to a Direct3DDevice8 object.

' For the projection matrix, set up a perspective transform, which
' transforms geometry from 3-D view space to 2-D viewport space,
' with a perspective divide that makes objects smaller in the
' distance. To build a perspective transform, you need the field
' of view (1/4 pi is common), the aspect ratio, and the near and
' far clipping planes, which define the distances at which
' geometry should be no longer be rendered.
```

```
Dim matProj As D3DMATRIX

D3DXMatrixPerspectiveFovLH matProj, g_pi / 4, 1, 1, 1000
g_D3DDevice.SetTransform D3DTS_PROJECTION, matProj
```

After you create the matrix, you must set it in a call to the **Direct3DDevice8.SetTransform** method, specifying D3DTRANSFORMSTATE_PROJECTION in the first parameter. For details, see Setting Transformations.

A W-Friendly Projection Matrix

Microsoft® Direct3D® can use the W component of a vertex that has been transformed by the world, view, and projection matrices to perform depth-based calculations in depth-buffer or fog effects. Computations such as these require that your projection matrix normalize W to be equivalent to world-space Z. In short, if your projection matrix includes a (3,4) coefficient that is not 1, you must scale all the coefficients by the inverse of the (3,4) coefficient to make a proper matrix. If you don't provide a compliant matrix, fog effects and depth buffering are not applied correctly. The projection matrix recommended in What Is the Projection Transformation? is compliant with w-based calculations.

The following illustration shows a non-compliant projection matrix, and the same matrix scaled so that eye-relative fog will be enabled.

Non-compliant

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & e \\ 0 & 0 & d & 0 \end{bmatrix}$$

Compliant

$$\begin{bmatrix} a/e & 0 & 0 & 0 \\ 0 & b/e & 0 & 0 \\ 0 & 0 & c/e & 1 \\ 0 & 0 & d/e & 0 \end{bmatrix}$$

In the preceding matrices, all variables are assumed to be nonzero. For more information about eye-relative fog, see Eye-Relative vs. Z-based Depth. For information about w-based depth buffering, see What Are Depth Buffers?

Note

Direct3D uses the currently set projection matrix in its w-based depth calculations. As a result, applications must set a compliant projection matrix to receive the desired w-based features, even if they do not use the Direct3D transformation pipeline.

The Lighting Engine

This section introduces the general concepts behind the Microsoft® Direct3D® lighting engine. The following topics are discussed.

- Enabling and Disabling the Lighting Engine
- Light Color Types and Sources
- Fog Effects

For detailed information on the inner working of the lighting engine, see Mathematics of Direct3D Lighting.

Enabling and Disabling the Lighting Engine

[C++]
By default, Microsoft® Direct3D® performs lighting calculations on all vertices, even those without vertex normals. This is different from the behavior in previous releases of Microsoft DirectX®, where lighting was performed only on vertices that contained a vertex normal. However, you can disable lighting through the D3DRS_LIGHTING render state. Call the **IDirect3DDevice8::SetRenderState** method, passing D3DRS_LIGHTING as the first parameter, and TRUE or FALSE as the second parameter. Setting the state to TRUE enables lighting (the default), and setting it to FALSE disables lighting operations.

[Visual Basic]
By default, Microsoft® Direct3D® performs lighting calculations on all vertices, even those without vertex normals. However, you can disable lighting through the D3DRS_LIGHTING render state. Call the **Direct3DDevice8.SetRenderState** method, passing D3DRS_LIGHTING as the first parameter, and True or False as the second parameter. Setting the state to True enables lighting (the default), and setting it to False disables lighting operations.

Light Color Types and Sources

Light sources in Microsoft® Direct3D® emit diffuse, ambient, and specular colors as distinct light components that factor into lighting computations independently of each other. Similarly, the vertices that the system renders can use discrete diffuse, ambient, specular, and emissive colors, as defined by their vertex format.

[C++]
For a C++ application, these vertex colors are used by the system only when you set the D3DRS_COLORVERTEX render state to TRUE. Vertex color sources are selectable; that is, you can configure the system to select the source for the vertex-color portions of lighting formulas at run time. The D3DRS_AMBIENTMATERIALSOURCE, D3DRS_DIFFUSEMATERIALSOURCE, and D3DRS_SPECULARMATERIALSOURCE render states control the source from which the system draws the associated colors during lighting calculations. You can

set each render state to a member of the **D3DMATERIALCOLORSOURCE** enumerated type, which defines values that cause the system to use the current material, or a color from the vertex, as the color source.

[Visual Basic]
From Microsoft® Visual Basic®, vertex colors are used by the system only when you set the D3DRS_COLORVERTEX render state to True. Vertex color sources are selectable; that is, you can configure the system to select the source for the vertex-color portions of lighting formulas at run time. The D3DRS_AMBIENTMATERIALSOURCE, D3DRS_DIFFUSEMATERIALSOURCE, and D3DRS_SPECULARMATERIALSOURCE render states control the source from which the system draws the associated colors during lighting calculations. You can set each render state to a member of the **CONST_D3DMATERIALCOLORSOURCE** enumeration, which defines values that cause the system to use the current material, or a color from the vertex, as the color source.

Fog Effects

The Microsoft® Direct3D® transformation and lighting engine can create fog effects during lighting. This type of fog is usually referred to as vertex fog because fog information is placed in the alpha component of the vertex to be rasterized.

For more information see Fog and Vertex Fog.

Mathematics of Direct3D Lighting

This section describes the mechanics and implementation details behind the Microsoft® Direct3D® lighting engine. Direct3D models illumination by estimating how light behaves in nature. The Direct3D light model keeps track of light color, the direction and distance that light travels, the position of the viewer, and the characteristics of the current material to compute two color components for each vertex in a face. Direct3D uses these color components to compute the color it draws while rasterizing the pixels of a face.

Note

All computations are made in model space by transforming the light source's position and direction, along with the camera position to model space using the inverse of the world matrix, then they are back transformed. As a result, if the world or view matrices introduce nonuniform scaling, the resultant lighting might be inaccurate.

This section presents a technical look at the formulas that Direct3D uses to come up with diffuse and specular components. By understanding the approach of Direct3D, you will be better equipped to decide if the Direct3D light model suits your needs. The Direct3D light model was designed to be accurate, efficient, and easy to use.

However, if the formulas used by Direct3D don't suit your needs, you can implement your own light model, bypassing the Direct3D lighting module altogether. The following topics are discussed.

- Lighting Notation
- Camera Space Transformation
- Basic Lighting Formula
- Diffuse Formula
- Specular Formula
- Light Attenuation Over Distance
- Reflectance Model
- Spotlight Falloff Model

The parameters used for these formulas are listed in the tables below. Each table has its corresponding default value, type, and a range of accepted values.

The following table lists the position parameters.

Parameter	Default value	Type	Description
Pe	(0,0,0)	D3DVECTOR	Camera position in camera space.
Po	N/A	D3DVECTOR	Position of current model origin (0,0,0,1) the camera space. This is the fourth row o the **Mw*Mv** matrix.
Mw	N/A	D3DMATRIX	World matrix, set by D3DTRANSFORMSTATE_VIEW.
Mv	N/A	D3DMATRIX	View matrix, set by D3DTRANSFORMSTATE_VIEW.
Mwv	N/A	D3DMATRIX	**Mw*Mv.**
Halfway	N/A	D3DVECTOR	Normalized vector, used to compute specular reflection.
La	(0.0, 0.0, 0.0, 0.0)	D3DCOLORVALUE	Ambient color in the light state. Set by D3DRENDERSTATE_AMBIENT.
V	N/A	D3DVECTOR	Vertex position in camera space.
N	N/A	D3DVECTOR	Normalized vertex normal in camera spac
Vcd	N/A	D3DCOLORVALUE	Vertex diffuse color.
Vcs	N/A	D3DCOLORVALUE	Vertex specular color.

The following table lists the material parameters.

Parameter	Default value	Type	Description
Ma	(0.0, 0.0, 0.0, 0.0)	D3DCOLORVALUE	Ambient color.
Md	(255, 255, 255, 255)	D3DCOLORVALUE	Diffuse color.

Ms	(0.0, 0.0, 0.0, 0.0)	D3DCOLORVALUE	Specular color.
Me	(0.0, 0.0, 0.0, 0.0)	D3DCOLORVALUE	Emissive color.
Mp	0.0	D3DVALUE	Specular exponent. Range: (-∞,+∞)

The following table lists the light source *i* parameters.

Parameter	Default value	Type	Description
Lp_I	(0.0, 0.0, 0.0)	D3DVECTOR	Position in camera space.
Ld_I	(0.0, 0.0, 1.0)	D3DVECTOR	Direction to the light in the camera space.
Lr_I	0.0	D3DVALUE	Distance range. Range: [0.0, D3DLIGHT_RANGE_MAX]
Lca_I	(0.0, 0.0, 0.0, 0.0)	D3DCOLORVALUE	Ambient color.
Lcs_I	(0.0, 0.0, 0.0, 0.0)	D3DCOLORVALUE	Specular color.
Lc_I	(1.0, 1.0, 1.0, 0.0)	D3DCOLORVALUE	Diffuse color.
$att0_I$	0.0	D3DVALUE	Constant attenuation factor. Range: (0, +∞
$att1_I$	0.0	D3DVALUE	Linear attenuation factor. Range: (0, +∞)
$att2_I$	0.0	D3DVALUE	Quadratic attenuation factor. Range: (0, +
$falloff_I$	0.0	D3DVALUE	Falloff factor. Range: (-∞, +∞)
$theta_I$	0.0	D3DVALUE	Umbra angle of spotlight in radians. Rang [0, π)
phi_I	0.0	D3DVALUE	Penumbra angle of spotlight in radians. Range: [$theta_I$, π)

Lighting Notation

[C++]

The following notations are used in the lighting formulas.

The range of a **D3DCOLORVALUE** component is (-∞, +∞).

• - dot product is defined as **d1** • **d2** = max{**d1**•**d2**, 0}

norm(**P**) - normalized vector

$\mathbf{V_1V_2}$ - vector from point $\mathbf{V_1}$ to point $\mathbf{V_2}$

$\mathbf{M}^{-1}$ - inverse matrix

$\mathbf{M}^{T}$ - transposed matrix

V1 / V2 , where **V1** and **V2** are of **D3DVECTOR** type, equals to the vector: (V1.x / V2.x, V1.y / V2.y , V1.z / V2.z).

V1 * V2 , where **V1** and **V2** are of **D3DVECTOR** type, equals to the vector: (V1.x * V2.x, V1.y * V2.y , V1.z * V2.z).

[Visual Basic]
The following notations are used in the lighting formulas.

The range of a **D3DCOLORVALUE** component is (-∞, +∞).

• - dot product is defined as **d1** • **d2** = max{**d1**•**d2**, 0}

norm(**P**) - normalized vector

$\mathbf{V_1V_2}$ - vector from point $\mathbf{V_1}$ to point $\mathbf{V_2}$

$\mathbf{M}^{-1}$ - inverse matrix

$\mathbf{M}^{T}$ - transposed matrix

V1 / V2 , where **V1** and **V2** are of **D3DVECTOR** type, equals to the vector: (V1.x / V2.x, V1.y / V2.y , V1.z / V2.z).

V1 * V2 , where **V1** and **V2** are of **D3DVECTOR** type, equals to the vector: (V1.x * V2.x, V1.y * V2.y , V1.z * V2.z).

Camera Space Transformation

Vertices in the camera space are computed by multiplying the object vertices by **Mwv** matrix.

$v = V_{object} * \mathbf{Mwv}$

Normals in the camera space are computed by multiplying the object normals by inverse transposed **Mwv** matrix and normalizing the result.

$N = N_{object} * (\mathbf{Mwv}^{-1})^{T}$

If D3DRENDERSTATE_NORMALIZENORMALS is set to TRUE, normals are normalized after transformation to the camera space:

N = **norm**(N)

Light position in the camera space ($\mathbf{Lp_i}$) is computed by multiplying the light source position by **Mv**.

$Lp_i = Lp_{original} * \mathbf{Mv}$

Direction to light in the camera space for D3DLIGHT_DIRECTIONAL lights is computed by multiplying the light source position by **Mv** matrix, normalizing and negating the result.

$Ld_i = -\mathbf{norm}(Ld_{i\ original} * \mathbf{Mv})$

For the D3DLIGHT_POINT and D3DLIGHT_SPOT the direction to light is computed as:

Ld_i = **norm**(VLp_i)

Basic Lighting Formula

The output of the lighting stage is diffuse (D) and specular (S) colors in RGBA format. Lighting may be in one of two states:

1. Lighting Off (D3DRENDERSTATE_LIGHTING is set to FALSE). In this state the vertex color is computed as follows:

D3DRENDERSTATE_COLORVETEX is ignored.

If the diffuse vertex color is present, the output diffuse color is equal to the vertex diffuse color. Otherwise, the diffuse color is equal to the default diffuse color (255,255,255,255). Output diffuse color is scaled and clamped to the range [0, 255].

If the specular vertex color is present, the output specular color is equal to the vertex specular color. Otherwise, the specular color is equal to the default specular (0,0,0,0). Output specular color is scaled and clamped to the range [0, 255].

2. Lighting On (D3DRENDERSTATE_LIGHTING is set to TRUE). In this state, Direct3D computes vertex colors according to the formulas below.

If normals are not present in the vertices, the part of the lighting equations that depends on dot product is set to zero. Lighting is still computed.

Lighting is done in the camera space.

alpha component is not used in the following lighting equations.

Diffuse Formula

The following explains the formula for diffuse lighting.

The specular component is set to (0, 0, 0, 0) if D3DRENDERSTATE_SPECULARENABLE is set to FALSE, otherwise it is computed as follows:

where

Specular Formula

The following explains the formula for specular lighting.

$\mathbf{h}_i$ are half way vectors between the normal and the direction to light.

$\mathbf{h}_i = \mathbf{norm}(\mathbf{norm}(\mathbf{Vpe}) + \mathbf{Ld}_i))$, if D3DRENDERSTATE_LOCALVIEWER = TRUE

$\mathbf{h}_i = \mathbf{norm}((0,0,-1) + \mathbf{Ld}_i))$, if D3DRENDERSTATE_LOCALVIEWER = FALSE

$rho_i = \mathbf{norm}(\mathbf{Ld}_i) \bullet \mathbf{norm}(\mathbf{VLp}_i)$

d_i is a distance from a vertex to the light *i* and is computed as follows:

Diffuse and specular components are clamped to be from 0 to 255, after all lights are processed and interpolated separately.

Light Attenuation Over Distance

[C++]

Microsoft® Direct3D® determines the distance between a light source and a vertex being lit by taking the magnitude of the vector that exists between the light's position and the vertex. This is represented by the following formula.

$$D = \left\| \mathrm{VL} \right\|$$

In the preceding formula, *D* is the distance being calculated, *V* is the position of the vertex being lit, and *L* is the light source's position. If *D* is greater than the light's range, that is, the **Range** member of a **D3DLIGHT8** structure, Direct3D makes no further attenuation calculations and applies no effects from the light to the vertex. If the distance is within the light's range, Direct3D then applies the following formula to

calculate light attenuation over distance for point lights and spotlights; directional lights don't attenuate.

$$A = \frac{1}{dvAttenuation0 + D \times dvAttenuation1 + D^2 \times dvAttenuation2}$$

In this attenuation formula, *A* is the calculated total attenuation and *D* is the distance from the light source to the vertex. The *dvAttenuation0*, *dvAttenuation1*, and *dvAttenuation2* values are the light's attenuation constants as specified by the members of a light object's **D3DLIGHT8** structure. The corresponding structure members are **Attenuation0**, **Attenuation1**, and **Attenuation2**.

The attenuation constants act as coefficients in the formula—you can produce a variety of attenuation curves by making simple adjustments to them. You can set **Attenuation0** to 1.0 to create a light that doesn't attenuate but is still limited by range, or you can experiment with different values to achieve various attenuation effects.

The attenuation at the maximum range of the light is not 0.0. To prevent lights from suddenly appearing when they are at the light range, an application can increase the light range. Or, the application can set up attenuation constants so that the attenuation factor is close to 0.0 at the light range. The attenuation value is multiplied by the red, green, and blue components of the light's color to scale the light's intensity as a factor of the distance light travels to a vertex.

After computing the light attenuation, Direct3D also considers spotlight effects if applicable, the angle that the light reflects from a surface, and the reflectance of the current material to calculate the diffuse and specular components for that vertex. For more information, see Spotlight Falloff Model and Reflectance Model.

[Visual Basic]
Microsoft® Direct3D® determines the distance between a light source and a vertex being lit by taking the magnitude of the vector that exists between the light's position and the vertex. This is represented by the following formula.

$$D = \left\| VL \right\|$$

In the preceding formula, *D* is the distance being calculated, *V* is the position of the vertex being lit, and *L* is the light source's position. If *D* is greater than the light's range—the **Range** member of a **D3DLIGHT8** type—Direct3D makes no further attenuation calculations and applies no effects from the light to the vertex. If the distance is within the light's range, Direct3D then applies the following formula to calculate light attenuation over distance for point lights and spotlights; directional lights don't attenuate.

$$A = \frac{1}{attenuation0 + D \times attenuation1 + D^2 \times attenuation2}$$

In this attenuation formula, *A* is the calculated total attenuation and *D* is the distance from the light source to the vertex. The *attenuation0*, *attenuation1*, and *attenuation2* values are the light's attenuation constants as specified by the members of a light object's **D3DLIGHT8** type. The corresponding structure members are **Attenuation0**, **Attenuation1**, and **Attenuation2**.

The attenuation constants act as coefficients in the formula—you can produce a variety of attenuation curves by making simple adjustments to them. You can set **Attenuation0** to 1.0 to create a light that doesn't attenuate but is still limited by range, or you can experiment with different values to achieve various attenuation effects.

The attenuation at the maximum range of the light is not 0.0. To prevent lights from suddenly appearing when they are at the light range, an application can increase the light range. Or, the application can set up attenuation constants so that the attenuation factor is close to 0.0 at the light range. The attenuation value is multiplied by the red, green, and blue components of the light's color to scale the light's intensity as a factor of the distance light travels to a vertex.

After computing the light attenuation, Direct3D also considers spotlight effects if applicable, the angle that the light reflects from a surface, and the reflectance of the current material to calculate the diffuse and specular components for that vertex. For more information, see Spotlight Falloff Model and Reflectance Model.

Reflectance Model

After adjusting the light intensity for any attenuation effects, Microsoft® Direct3D® computes how much of the remaining light reflects from a vertex given the angle of the vertex normal and the direction of the incident light. Direct3D skips to this step for directional lights because they don't attenuate over distance.

The system considers two reflection types, diffuse and specular, and uses a different formula to determine how much light is reflected for each. After calculating the amounts of light reflected, Direct3D applies these new values to the diffuse and specular reflectance properties of the current material. The resulting color values are the diffuse and specular components that the rasterizer uses to produce Gouraud shading and specular highlighting.

Diffuse Reflection Model

[C++]
Microsoft Direct3D uses the following formula to compute diffuse reflection factors.

$$R_d = -D \bullet N$$

In this formula, R_d is the diffuse reflectance factor, D is the direction that the light travels to the vertex, and N is the vertex normal. Vector D is normalized; vector N is normalized only if the D3DRS_NORMALIZENORMALS render state is enabled. The light's direction vector is reversed by multiplying it by -1 to create the proper association between the direction vector and the vertex normal. This formula produces values that range from -1.0 to 1.0, which are clamped to the range of 0.0 to 1.0 and used to scale the intensity of the light reflecting from the vertex.

After the diffuse reflection formula is applied, the scaled light is then applied to the diffuse reflectance formula to determine the diffuse component at that vertex. The formula that combines ambient and diffuse reflection to create the diffuse component for the vertex looks like this:

$$D_v = I_a V_a + V_e + \sum_i A(R_d V_d L_d + V_a L_a)$$

In the preceding formula, D_v is the diffuse component being calculated for the vertex, I_a is the ambient light level in the scene, and A is the light intensity for a light source that has been attenuated for distance and spotlight effects (attenuation from Light Attenuation Over Distance multiplied by Spotlight Falloff Model). The L variables represent the light's properties, and the V entries represent the vertex color, where the subscripts a, d, and e applied to each denote the type of color—ambient, diffuse, or emissive. As the formula notation states, the system computes $I_aV_a + V_e$ once, adding $A(R_dV_dL_d + V_aL_a)$ for every active light.

If the D3DRS_COLORVERTEX render state is enabled, the system selects colors for V based on the values set for the D3DRS_AMBIENTMATERIALSOURCE and D3DRS_DIFFUSEMATERIALSOURCE render states. Set these render states to a member of the **D3DMATERIALCOLORSOURCE** enumerated type to cause the system to use the current material, or a color from the vertex, as the color source.

For more information, see Specular Reflection Model.

[Visual Basic]
Microsoft Direct3D uses the following formula to compute diffuse reflection factors.

$$R_d = -D \bullet N$$

In this formula, R_d is the diffuse reflectance factor, D is the direction that the light travels to the vertex, and N is the vertex normal. Vector D is normalized; vector N is normalized only if the D3DRS_NORMALIZENORMALS render state is enabled. The light's direction vector is reversed by multiplying it by -1 to create the proper association between the direction vector and the vertex normal. This formula produces values that range from -1.0 to 1.0, which are clamped to the range of 0.0 to 1.0 and used to scale the intensity of the light reflecting from the vertex.

After the diffuse reflection formula is applied, the scaled light is then applied to the diffuse reflectance formula to determine the diffuse component at that vertex. The formula that combines ambient and diffuse reflection to create the diffuse component for the vertex looks like this:

$$D_v = I_a V_a + V_e + \sum_i A(R_d V_d L_d + V_a L_a)$$

In the preceding formula, D_v is the diffuse component being calculated for the vertex, I_a is the ambient light level in the scene, and A is the light intensity for a light source that has been attenuated for distance and spotlight effects—attenuation from Light Attenuation Over Distance multiplied by Spotlight Falloff Model. The L variables represent the light's properties, and the V entries represent the vertex color, where the subscripts a, d, and e applied to each denote the type of color—ambient, diffuse, or emissive. As the formula notation states, the system computes $I_aV_a + V_e$ once, adding $A(R_dV_dL_d + V_aL_a)$ for every active light.

If the D3DRS_COLORVERTEX render state is enabled, the system selects colors for V based on the values set for the D3DRS_AMBIENTMATERIALSOURCE and D3DRS_DIFFUSEMATERIALSOURCE render states. Set these render states to a member of the **CONST_D3DMATERIALCOLORSOURCE** enumeration to cause the system to use the current material, or a color from the vertex, as the color source.

For more information, see Specular Reflection Model.

Specular Reflection Model

[C++]
Modeling specular reflection requires that the system not only know the direction that light is traveling, but also the direction to the viewer's eye. The system uses a simplified version of the Phong specular-reflection model, which employs a *halfway vector* to approximate the intensity of specular reflection. This halfway vector exists midway between the vector to the light source and the vector to the eye. Microsoft Direct3D provides applications with two ways to compute the halfway vector, which is controlled by the D3DRS_LOCALVIEWER render state. If D3DRS_LOCALVIEWER is set to TRUE, the system calculates the halfway vector using the position of the camera and the position of the vertex, along with the light's direction vector. The following formula illustrates this.

$$H = \text{norm}\left(\text{norm}\left(VC\right) - L_d\right)$$

In the preceding formula, *norm* is an operator that normalizes an input vector, *VC* is the vector that exists from the position of the vertex to the position of the viewpoint or eye, and L_d is the light's direction vector.

Determining the halfway vector in this manner can be computationally expensive. As an alternative, you can set D3DRS_LOCALVIEWER to FALSE. This instructs the system to act as though the viewpoint is infinitely distant on the z-axis. This setting is less computationally expensive, but much less accurate, so it is best used by applications that use orthogonal projection. When D3DRS_LOCALVIEWER is set to FALSE, Direct3D determines the halfway vector by the following formula.

$$H = \text{norm}\ (I - L_d)$$

This formula is similar to the first formula, but substitutes the vector *I* (0, 0, -1)—which points at a viewpoint infinitely distant on the z-axis—instead of computing the vector *VC*.

After determining the halfway vector, *H*, the system uses the following formula to compute specular reflection.

$$R_s = (N \bullet H)^p$$

In the preceding formula, R_s is the specular reflectance, *N* in the vertex normal, *H* is the halfway vector, and *p* is the specular reflection power of the current material, as specified by the **Power** member of the material's **D3DMATERIAL8** structure. Vector *H* is normalized, and vector *N* is normalized only if the D3DRS_NORMALIZENORMALS render state is enabled.

As with the diffuse reflectance formula, this formula produces values that range from -1.0 to 1.0, which are clamped to the range of 0.0 to 1.0 and used to scale the light reflecting from the vertex. Also similar to the diffuse reflection model, the remaining light is applied to a formula that derives the specular component at that vertex:

$$S_v = V_s A\ R_s L_s$$

In this formula, S_v is the specular color being computed. *A* is the light from a single light source that has been attenuated for distance and spotlight effects; for more information, see Light Attenuation Over Distance and Spotlight Falloff Model. The R_s variable is the previously calculated specular reflectance, V_s is the specular component for the vertex, and L_s is the specular light color output by the light.

If the D3DRS_COLORVERTEX render state is enabled, the system selects the color source for *V* based on the value of the D3DRS_SPECULARMATERIALSOURCE render state. This render state can be set to a member of the

D3DMATERIALCOLORSOURCE enumerated type to cause the system to use the current material or one of the color components for the vertex as the color source.

For more information, see Diffuse Reflection Model.

[Visual Basic]
Modeling specular reflection requires that the system not only know the direction that light is traveling, but also the direction to the viewer's eye. The system uses a simplified version of the Phong specular-reflection model, which employs a *halfway vector* to approximate the intensity of specular reflection. This halfway vector exists midway between the vector to the light source and the vector to the eye. Microsoft® Direct3D® provides applications with two ways to compute the halfway vector, which is controlled by the D3DRS_LOCALVIEWER render state. If D3DRS_LOCALVIEWER is set to True, the system calculates the halfway vector using the position of the camera and the position of the vertex, along with the light's direction vector. The following formula illustrates this.

$$H = \text{norm}\left(\text{norm}\left(VC\right) - L_d\right)$$

In the preceding formula, *norm* is an operator that normalizes an input vector, *VC* is the vector that exists from the position of the vertex to the position of the viewpoint or eye, and L_d is the light's direction vector.

Determining the halfway vector in this manner can be computationally intensive. As an alternative, you can set D3DRS_LOCALVIEWER to FALSE. This instructs the system to act as though the viewpoint is infinitely distant on the z-axis. This setting is less computationally expensive, but much less accurate, so it is best used by applications that use orthogonal projection. When D3DS_LOCALVIEWER is set to False, Direct3D determines the halfway vector by the following formula.

$$H = \text{norm}\left(I - L_d\right)$$

This formula is similar to the first formula, but substitutes the vector $I(0, 0, -1)$—which points at a viewpoint infinitely distant on the z-axis—instead of computing the vector *VC*.

After determining the halfway vector, *H*, the system uses the following formula to compute specular reflection.

$$R_s = (N \bullet H)^p$$

In the preceding formula, R_s is the specular reflectance, N in the vertex normal, H is the halfway vector, and p is the specular reflection power of the current material as specified by the **power** member of the material's **D3DMATERIAL8** type. Vector H is normalized, and vector N is normalized only if the D3DRS_NORMALIZENORMALS render state is enabled.

As with the diffuse reflectance formula, this formula produces values that range from -1.0 to 1.0, which are clamped to the range of 0.0 to 1.0 and used to scale the light reflecting from the vertex. Also similar to the diffuse reflection model, the remaining light is applied to a formula that derives the specular component at that vertex:

$$S_v = V_s A R_s L_s$$

In the preceding formula, S_v is the specular color being computed. A is the light from a single light source that has been attenuated for distance and spotlight effects; for more information, see Light Attenuation Over Distance and Spotlight Falloff Model. The R_s variable is the previously calculated specular reflectance, V_s is the selected specular component for the vertex, and L_s is the specular light color output by the light.

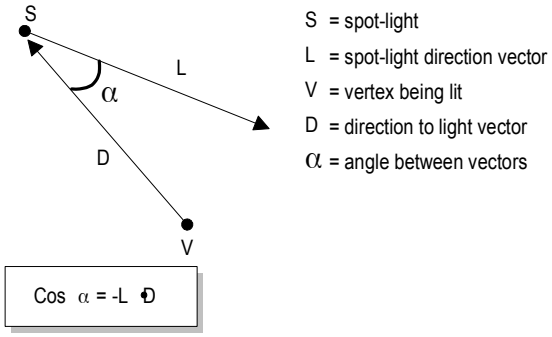
If the D3DRS_COLORVERTEX render state is enabled, the system selects the color source for V based on the value of the D3DRS_SPECULARMATERIALSOURCE render state. This render state can be set to a member of the **CONST_D3DMATERIALCOLORSOURCE** enumerated type to cause the system to use the current material or one of the color components for the vertex as the color source.

For more information, see Diffuse Reflection Model.

Spotlight Falloff Model

[C++]
Spotlights emit a cone of light that has two parts: a bright inner cone and an outer cone. Light is brightest in the inner cone and isn't present outside the outer cone, with light intensity attenuating between the two areas. This type of attenuation is commonly referred to as falloff.

How much light a vertex receives is based on the vertex's location in the inner or outer cones. Microsoft® Direct3D® computes the dot product of the spotlight's direction vector (L) and the vector from the vertex to the light (D). This value is equal to the cosine of the angle between the two vectors, and serves as an indicator of the vertex's position that can be compared to the light's cone angles to determine where the vertex might lie in the inner or outer cones. The following illustration provides a graphical representation of the association between these two vectors.

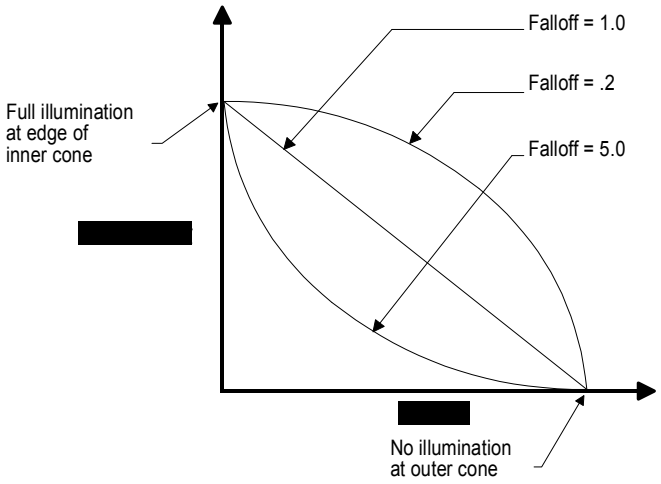


Next, the system compares this value to the cosine of the spotlight's inner and outer cone angles. In the light's **D3DLIGHT8** structure, the **Theta** and **Phi** members represent the total cone angles for the inner and outer cones. Because the attenuation occurs as the vertex becomes more distant from the center of illumination, rather than across the total cone angle, Direct3D halves these cone angles before calculating their cosines.

If the dot product of vectors *L* and *D* is less than or equal to the cosine of the outer cone angle, the vertex lies beyond the outer cone and receives no light. If the dot product of *L* and *D* is greater than the cosine of the inner cone angle, then the vertex is within the inner cone and receives the maximum amount of light, still considering attenuation over distance. If the vertex is somewhere between the two regions, Direct3D calculates falloff for the vertex by using the following formula.

$$I_f = \left(\frac{Cos\alpha - Cos\phi}{Cos\theta - Cos\phi} \right)^p$$

In the formula, I_f is light intensity, after falloff, for the vertex being lit, α is the angle between vectors *L* and *D*, ϕ is half of the outer cone angle, θ is half of the inner cone angle, and *p* is the spotlight's falloff property—**Falloff** in the **D3DLIGHT8** structure. This formula generates a value between 0.0 and 1.0 that scales the light's intensity at the vertex to account for falloff. Attenuation as a factor of the vertex's distance from the light is also applied. The *p* value corresponds to the **Falloff** member of the **D3DLIGHT8** structure and controls the shape of the falloff curve. The following illustration shows how different **Falloff** values can affect the falloff curve.

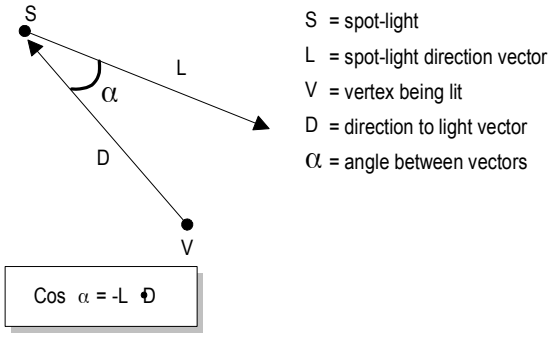


The effect of various **Falloff** values on the actual lighting is subtle, and a small performance penalty is incurred by shaping the falloff curve with **Falloff** values other than 1.0. For these reasons, this value is typically set to 1.0.

For more information, see Light Attenuation Over Distance.

[Visual Basic]
Spotlights emit a cone of light that has two parts: a bright inner cone and an outer cone. Light is brightest in the inner cone and isn't present outside the outer cone, with light intensity attenuating between the two areas. This type of attenuation is commonly referred to as falloff.

How much light a vertex receives is based on the vertex's location in the inner or outer cones. Microsoft® Direct3D® computes the dot product of the spotlight's direction vector (*L*) and the vector from the vertex to the light (*D*). This value is equal to the cosine of the angle between the two vectors, and serves as an indicator of the vertex's position that can be compared to the light's cone angles to determine where the vertex might lie in the inner or outer cones. The following illustration provides a graphical representation of the association between these two vectors.

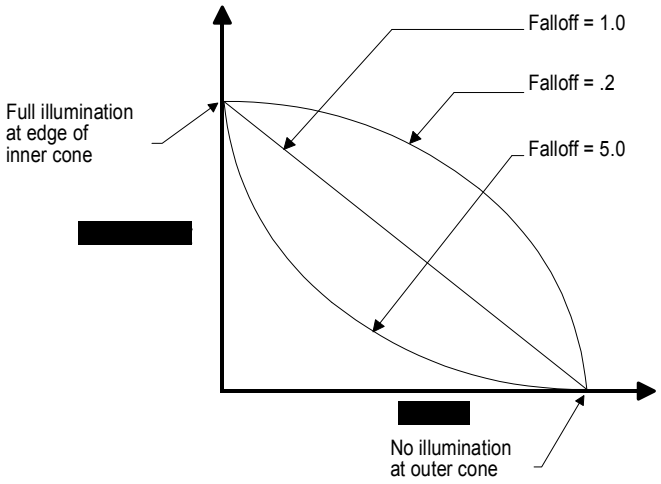


Next, the system compares this value to the cosine of the spotlight's inner and outer cone angles. In the light's **D3DLIGHT8** type, the **Theta** and **Phi** members represent the total cone angles for the inner and outer cones. Because the attenuation occurs as the vertex becomes more distant from the center of illumination, rather than across the total cone angle, Direct3D halves these cone angles before calculating their cosines.

If the dot product of vectors *L* and *D* is less than or equal to the cosine of the outer cone angle, the vertex lies beyond the outer cone and receives no light. If the dot product of *L* and *D* is greater than the cosine of the inner cone angle, then the vertex is within the inner cone and receives the maximum amount of light, still considering attenuation over distance. If the vertex is somewhere between the two regions, Direct3D calculates falloff for the vertex by using the following formula.

$$I_f = \left(\frac{Cos\alpha - Cos\phi}{Cos\theta - Cos\phi} \right)^p$$

In the formula, I_f is light intensity, after falloff, for the vertex being lit, α is the angle between vectors *L* and *D*, ϕ is half of the outer cone angle, θ is half of the inner cone angle, and *p* is the spotlight's falloff property—**Falloff** in the **D3DLIGHT8** type. This formula generates a value between 0.0 and 1.0 that scales the light's intensity at the vertex to account for falloff. Attenuation as a factor of the vertex's distance from the light is also applied. The *p* value corresponds to the **Falloff** member of the **D3DLIGHT8** type and controls the shape of the falloff curve. The following illustration shows how different **Falloff** values can affect the falloff curve.



The effect of various **Falloff** values on the actual lighting is subtle, and a small performance penalty is incurred by shaping the falloff curve with values other than 1.0. For these reasons, this value is typically set to 1.0.

For more information, see Light Attenuation Over Distance.

Viewports and Clipping

This section discusses clipping, the last stage of the geometry pipeline. The discussion is organized into the following topics.

- What Is a Viewport?
- Viewport Rectangle
- Clipping Volumes
- Viewport Scaling
- Using Viewports

Microsoft® Direct3D® implements clipping by way of a set of viewport parameters set in the device.

What Is a Viewport?

Conceptually, a viewport is a 2-D rectangle into which a three-dimensional scene is projected. In Microsoft® Direct3D®, the rectangle exists as coordinates within a Direct3D surface that the system uses as a rendering target. The projection transformation converts vertices into the coordinate system used for the viewport.

You use a viewport in Direct3D to specify the following features in your application.

- The screen-space viewport to which the rendering will be confined.

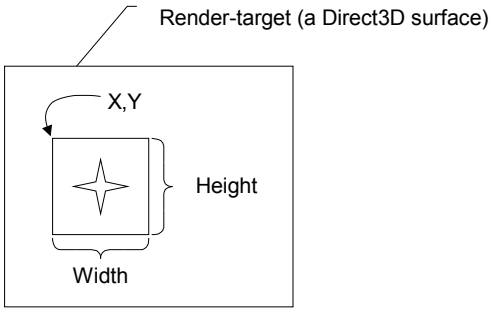
- The range of depth values on a render-target surface into which a scene will be rendered (usually 0.0 to 1.0).

Viewport Rectangle

[C++]
You define the viewport rectangle in C++ by using the **D3DVIEWPORT8** structure. The **D3DVIEWPORT8** structure is used with the following viewport manipulation methods exposed by the **IDirect3DDevice8** interface.

- **IDirect3DDevice8::GetViewport**
- **IDirect3DDevice8::SetViewport**

The **D3DVIEWPORT8** structure contains four members—**X**, **Y**, **Width**, and **Height**—that define the area of the render-target surface into which a scene will be rendered. These values correspond to the destination rectangle, or viewport rectangle, as shown in the following illustration.



The values you specify for the **X**, **Y**, **Width**, and **Height** members of the **D3DVIEWPORT8** structure are screen coordinates relative to the upper-left corner of the render-target surface. The structure defines two additional members (**MinZ** and **MaxZ**) that indicate the depth-ranges into which the scene will be rendered.

Microsoft® Direct3D® assumes that the viewport clipping volume ranges from -1.0 to 1.0 in X, and from 1.0 to -1.0 in Y. These were the settings used most often by applications in the past. During the projection transformation, you can adjust for viewport aspect ratio before clipping. This task is covered by topics in The Projection Transformation section.

Note

The **D3DVIEWPORT8** structure members **MinZ** and **MaxZ** indicate the depth-ranges into which the scene will be rendered and are not used for clipping. Most applications will set these members to 0.0 and 1.0 to enable the system to render to the entire range of depth values in the depth buffer. In some cases, you can achieve special effects by using other depth ranges. For instance, to render a heads-up display in a game, you can set both values to 0.0 to force the system to

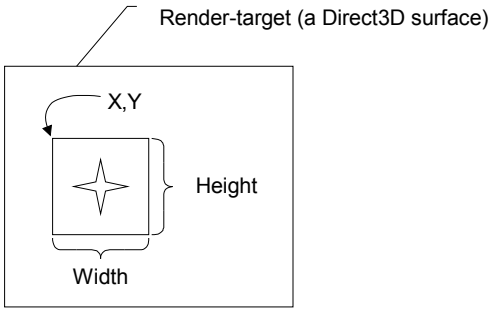
render objects in a scene in the foreground, or you might set them both to 1.0 to render an object that should always be in the background.

[Visual Basic]
You define the viewport rectangle in a Microsoft® Visual Basic® application by using the **D3DVIEWPORT8** type. The **D3DVIEWPORT8** type is used with the following viewport manipulation methods offered by the **Direct3DDevice8** class.

- **Direct3DDevice8.GetViewport**
- **Direct3DDevice8.SetViewport**

The **D3DVIEWPORT8** type contains four members—**x**, **y**, **Width**, and **Height**—that define the area of the render-target surface into which a scene will be rendered, called the viewport rectangle.

These values correspond to the destination rectangle, or viewport rectangle, as shown in the following illustration.



The values you specify for the **x**, **y**, **Width**, and **Height** members of the **D3DVIEWPORT8** type are screen coordinates relative to the upper-left corner of the render-target surface. The type defines two additional members (**MinZ** and **MaxZ**) indicate the depth-ranges into which the scene will be rendered.

Microsoft® Direct3D® assumes that the viewport clipping volume ranges from -1.0 to 1.0 in x, and from 1.0 to -1.0 in y. These were the settings used most often by applications in the past. In Microsoft DirectX® 8.0, as in previous releases of DirectX, you can adjust for viewport aspect ratio before clipping, during the projection transformation. This task is covered by topics in The Projection Transformation.

Clipping Volumes

The results of the projection matrix determine the clipping volume in projection space. Microsoft® Direct3D® defines the clipping volume in projection space as:

$$-W_c < X_c \le W_c$$
$$-W_c < Y_c \le W_c$$
$$0 < Z_c \le W_c$$

In the preceding formulas, X_c, Y_c, Z_c, and W_c represent the vertex coordinates after the projection transformation is applied. Any vertices that have an x, y, or z component outside these ranges are clipped, if clipping is enabled (the default behavior).

[C++]
With the exception of vertex buffers, applications enable or disable clipping by way of the D3DRS_CLIPPING render state. Clipping information for vertex buffers is generated during processing, for more information see Processing Vertices.

Direct3D does not clip transformed vertices of a primitive from a vertex buffer unless it comes from **IDirect3DDevice8::ProcessVertices**. If you are doing your own transforms and need Direct3D to do the clipping you should not use vertex buffers; in this case, the application traverses the data to transform it, Direct3D traverses the data a second time to clip it, and then the driver renders the data which is inefficient. So, if the application transforms the data is should also clip the data.

[Visual Basic]
With the exception of vertex buffers, applications enable or disable clipping by way of the D3DRS_CLIPPING render state. Clipping information for vertex buffers is generated during processing, for more information see Processing Vertices.

Direct3D does not clip transformed vertices of a primitive from a vertex buffer unless it comes from **Direct3DDevice8.ProcessVertices**. If you are doing your own transforms and need Direct3D to do the clipping you should not use vertex buffers; in this case, the application traverses the data to transform it, Direct3D traverses the data a second time to clip it, and then the driver renders the data which is inefficient. So, if the application transforms the data is should also clip the data.

Viewport Scaling

[C++]
The dimensions used in the **X**, **Y**, **Width**, and **Height** members of the **D3DVIEWPORT8** structure for a viewport define the location and dimensions of the viewport on the render-target surface. These values are in screen coordinates, relative to the upper-left corner of the surface.

Microsoft® Direct3D® uses the viewport location and dimensions to scale the vertices to fit a rendered scene into the appropriate location on the target surface. Internally, Direct3D inserts these values into a matrix that is applied to each vertex:

$$\begin{bmatrix} dwWidth/2 & 0 & 0 & 0 \\ 0 & -dwHeight/2 & 0 & 0 \\ 0 & 0 & dvMaxZ - dvMinZ & 0 \\ dwX + dwWidth/2 & dwHeight/2 + dwY & dvMinz & 1 \end{bmatrix}$$

This matrix simply scales vertices according to the viewport dimensions and desired depth range and translates them to the appropriate location on the render-target surface. The matrix also flips the y-coordinate to reflect a screen origin at the top-left corner with y increasing downward. After this matrix is applied, vertices are still homogeneous—that is, they still exist as [x,y,z,w] vertices—and they must be converted to non-homogeneous coordinates before being sent to the rasterizer. This is performed by way of simple division, as discussed in Rasterization.

Note

The viewport scaling matrix incorporates the **MinZ** and **MaxZ** members of the **D3DVIEWPORT8** structure to scale vertices to fit the depth range [**MinZ**, **MaxZ**]. This represents different semantics from previous releases of Microsoft DirectX, in which these members were used for clipping.
For more information, see Viewport Rectangle and Clipping Volumes.
Applications typically set **MinZ** and **MaxZ** to 0.0 and 1.0 to cause the system to render to the entire depth range. However, you can use other values to achieve certain affects. You might set both values to 0.0 to force all objects into the foreground, or set both to 1.0 to render all objects into the background.

[Visual Basic]
The dimensions used in the **x**, **y**, **Width**, and **Height** members of the **D3DVIEWPORT8** type for a viewport define the location and dimensions of the viewport on the render-target surface. These values are in screen coordinates, relative to the upper-left corner of the surface.

Microsoft® Direct3D® uses the viewport location and dimensions to scale the vertices to fit a rendered scene into the appropriate location on the target surface. Internally, Direct3D inserts these values into a matrix that is applied to each vertex:

$$\begin{bmatrix} lWidth/2 & 0 & 0 & 0 \\ 0 & -lHeight/2 & 0 & 0 \\ 0 & 0 & maxz - minz & 0 \\ lX + lWidth/2 & lHeight/2 + lY & minz & 1 \end{bmatrix}$$

This matrix simply scales vertices according to the viewport dimensions and desired depth range and translates them to the appropriate location on the render-target surface. The matrix also flips the y-coordinate to reflect a screen origin at the top-left corner with y increasing downward. After this matrix is applied, vertices are still homogeneous—that is, they still exist as [x,y,z,w] vertices—and they must be converted to non-homogeneous coordinates before being sent to the rasterizer. This is performed by way of simple division, as discussed in Rasterization.

Using Viewports

This section provides details about working with viewports. Information is divided into the following topics.

- Setting the Viewport Clipping Volume
- Clearing a Viewport
- Manually Transforming Vertices

Setting the Viewport Clipping Volume

The only requirement for configuring the viewport parameters for a rendering device is to set the viewport's clipping volume. To do this, you initialize and set clipping values for the clipping volume and for the render-target surface. Viewports are commonly set up to render to the full area of the render-target surface, but this isn't a requirement.

[C++]
You can use the following settings for the members of the **D3DVIEWPORT8** structure to achieve this in C++.

```
D3DVIEWPORT8 viewData = { 0, 0, width, height, 0.0f, 1.0f };
```

After setting values in the **D3DVIEWPORT8** structure, apply the viewport parameters to the device by calling its **IDirect3DDevice8::SetViewport** method. The following code example shows what this call might look like.

```
HRESULT hr;

hr = pd3dDevice->SetViewport(&viewData);
if(FAILED(hr))
   return hr;
```

If the call succeeds, the viewport parameters are set and will take effect the next time a rendering method is called. To make changes to the viewport parameters, just update the values in the **D3DVIEWPORT8** structure and call **SetViewport** again.

Note

The **D3DVIEWPORT8** structure members **MinZ** and **MaxZ** indicate the depth-ranges into which the scene will be rendered and are not used for clipping. Most

applications set these members to 0.0 and 1.0 to enable the system to render to the entire range of depth values in the depth buffer. In some cases, you can achieve special effects by using other depth ranges. For instance, to render a heads-up display in a game, you can set both values to 0.0 to force the system to render objects in a scene in the foreground, or you might set them both to 1.0 to render an object that should always be in the background.

[Visual Basic]
The following Microsoft® Visual Basic® code creates settings in a **D3DVIEWPORT8** type to achieve this.

```
Dim viewData As D3DVIEWPORT8

With viewData
   .x = 0: .y = 0
   .Width = Width: .Height = Height
   .MinZ = 0#: .MaxZ = 1#
End With
```

After setting values in the **D3DVIEWPORT8** type, apply the viewport parameters to the device by calling its **Direct3DDevice8.SetViewport** method.

```
' The d3dDevice variable contains a valid reference to a
' Direct3DDevice8 object.
Call d3dDevice.SetViewport(viewData)
```

After the call, the viewport parameters are set and will take effect the next time a rendering method is called. To make changes to the viewport parameters, just update the values in the **D3DVIEWPORT8** type and call **SetViewport** again.

Clearing a Viewport

Clearing the viewport resets the contents of the viewport rectangle on the render-target surface as well as the rectangle in the depth and stencil buffer surfaces, if specified. Typically, you clear the viewport before rendering a new frame to ensure that graphics and other data is ready to accept new rendered objects without displaying artifacts.

[C++]
The **IDirect3DDevice8** interface offers C++ developers the **IDirect3DDevice8::Clear** method to clear the viewport. The method accepts one or more rectangles that define the areas on the surfaces being cleared. In cases where the scene being rendered includes motion throughout the entire viewport rectangle—in a first-person perspective game, for example—you might want to clear the entire viewport each frame. In this situation, you set the *Count* parameter to 1, and the *pRects* parameter to the address of a single rectangle that covers the entire viewport

area. If it is more convenient, you can set the *pRects* parameter to NULL and the *Count* parameter to 0 to indicate that the entire viewport rectangle should be cleared.

The **Clear** method is flexible, and it provides support for clearing stencil bits within a depth buffer. The *Flags* parameter accepts three flags that determine how it clears the render target and any associated depth or stencil buffers. If you include the D3DCLEAR_TARGET flag, the method clears the viewport using an arbitrary RGBA color that you provide in the *Color* parameter (not the material color). If you include the D3DCLEAR_ZBUFFER flag, the method clears the depth buffer to an arbitrary depth you specify in *Z*:. 0.0 is the closest distance, and 1.0 is the farthest. Including the D3DCLEAR_STENCIL flag causes the method to reset the stencil bits to the value you provide in the *Stencil* parameter. You can use integers that range from 0 to 2^n-1, where *n* is the stencil buffer bit depth.

Note

Microsoft® DirectX® 5.0 allowed background materials to have associated textures, making it possible to clear the viewport to a texture rather than a simple color. This feature was little used, and not particularly efficient. Interfaces for DirectX 6.0 and later do not accept texture handles, meaning that you can no longer clear the viewport to a texture. Rather, applications must now draw backgrounds manually. As a result, there is rarely a need to clear the viewport on the render-target surface. As long as your application clears the depth buffer, all pixels on the render-target surface will be overwritten anyway.

In some situations, you might be rendering only to small portions of the render target and depth buffer surfaces. The clear methods also enable you to clear multiple areas of your surfaces in a single call. Do this by setting the *Count* parameter to the number of rectangles you want cleared, and specify the address of the first rectangle in an array of rectangles in the *pRects* parameter.

[Visual Basic]

The **Direct3DDevice8** Microsoft® Visual Basic® class offers the **Direct3DDevice8.Clear** method to clear the viewport. The method accepts one or more rectangles that define the area or areas on the surfaces being cleared. In cases where the scene being rendered includes motion throughout the entire viewport rectangle—in a first-person perspective game, for example—you might want to clear the entire viewport each frame. In this situation, you set the *Count* parameter to 1, and the *ClearD3DRect* parameter to a single-element array of **D3DRECT** variables, where the first and only element describes a rectangle that covers the entire area of the render-target surface.

The **Clear** method is flexible, and it provides support for clearing stencil bits in a depth buffer. The *Flags* parameter accepts three flags that determine how it clears the render target and any associated depth or stencil buffers. If you include the D3DCLEAR_TARGET flag, the method clears the viewport using an arbitrary RGBA color that you provide in the *Color* parameter (not the material color). If you include the D3DCLEAR_ZBUFFER flag, the method clears the depth buffer to an

arbitrary depth you specify in *Z*:. 0.0 is the closest distance, and 1.0 is the farthest. Including the D3DCLEAR_STENCIL flag causes the method to reset the stencil bits to the value you provide in the *Stencil* parameter. You can use integers that range from 0 to 2^n-1, where *n* is the stencil buffer bit depth.

In some situations, you might only be rendering to small portions of the render target and depth buffer surfaces. The clear methods also enable you to clear multiple areas of your surfaces in a single call. Do this by setting the *Count* parameter to the number of rectangles you want cleared, and an array of rectangles in the *ClearD3DRect* parameter.

Manually Transforming Vertices

You can use three kinds of vertices in your Microsoft® Direct3D® application. Read Vertex Formats for more details on the vertex formats.

[C++]

Untransformed and Unlit Vertices
: Vertices that your application doesn't light or transform. Although you specify lighting parameters and transformation matrices, Direct3D computes these values.

Untransformed and Lit Vertices
: Vertices that your application lights but does not transform.

Transformed and Lit Vertices
: Vertices that your application both lights and transforms.

You can change from simple to complex vertex types by using vertex buffers. Vertex buffers are objects used to efficiently contain and process batches of vertices for rapid rendering, and are optimized to exploit processor-specific features. Use the **IDirect3DDevice8::ProcessVertices** method to perform vertex transformations. **ProcessVertices** accepts only untransformed vertices and can optionally light and clip vertices as well. Lighting is performed at the time you call the **ProcessVertices** methods, but clipping is performed at render time.

After processing the vertices, you can use special rendering methods to render the vertices, or you can access them directly by locking the vertex buffer memory. For more information about using vertex buffers, see Vertex Buffers.

[Visual Basic]

Untransformed and Unlit Vertices
: Vertices that your application doesn't light or transform. Although you specify lighting parameters and transformation matrices, Direct3D computes these values.

Untransformed and Lit Vertices
: Vertices that your application lights but does not transform.

Transformed and Lit Vertices

Vertices that your application both lights and transforms.

You can change from simple to complex vertex types by using vertex buffers. Vertex buffers are objects used to efficiently contain and process batches of vertices for rapid rendering, and are optimized to exploit processor-specific features. Use the **Direct3DDevice8.ProcessVertices** method to perform vertex transformations for you. **ProcessVertices** accepts only untransformed vertices and can optionally light and clip vertices as well. Lighting is performed at the time you call the **ProcessVertices** methods, but clipping is performed at render time.

After processing the vertices, you can use special rendering methods to render the vertices, or you can access them directly by locking the vertex buffer memory. For more information about using vertex buffers, see Vertex Buffers.

Rasterization

After passing through the Microsoft® Direct3D® geometry pipeline, vertices have been transformed, clipped, and scaled to fit in the viewport on the render-target surface, making them almost ready to send to the rasterizer to paint on the screen. However, the vertices are still homogeneous, and the rasterizer expects to receive vertices in terms of their x-, y-, and z-locations, as well as the reciprocal-of-homogeneous-w (RHW). Direct3D converts the homogeneous vertices to non-homogeneous vertices by dividing the x-, y-, and z-coordinates by the w-coordinate. Direct3D then produces an RHW value by inverting the w-coordinate, as in the following formulas.

$$X_s = x/w$$
$$Y_s = y/w$$
$$Z_s = z/w$$
$$RHW = 1/w$$

The resulting values are passed to the rasterizer for display. The rasterizer uses the x- and y-coordinates as the screen coordinates for the vertex, and it uses the z-coordinate for depth comparisons in the depth buffer when z-buffering is enabled. The RHW value is used in multiple ways: for calculating fog, for performing perspective-correct texture mapping, and for w-buffering—an alternate form of depth buffering.

Programmable Vertex and Pixel Processing

Microsoft® DirectX® 8.0 features a new kind of graphics pipeline, featuring a high degree of programmability. The following diagram illustrates the major components

and data flow in the programmable vertex and pixel pipeline. The transformation and lighting engine and DirectX 6.0 and 7.0 multitexturing modules have been replaced in the programmable pipeline by vertex and pixel shaders.

The first new part of the pipeline is the high-order primitive module, which works to tessellate high-order primitives such as bézier and B-splines. The vertex shader is a programmable module used to execute geometric operations on the vertex data. The vertex shader can perform a variety of functions which includes standard transformation and lighting. The pixel shader module is a programmable pixel processor. Pixel shaders control the color and alpha blending operations and the texture addressing operations.

For general information on programmable shaders, see Introduction to Procedural Shaders.

For information on how the Microsoft Direct3D® pipeline for DirectX 8.0 compares to previous versions, see Integration of Vertex Shaders into the Geometry Pipeline.

For information on the nonprogrammable sections of the pipeline, which have remained unchanged from DirectX 6.0 and 7.0, see Viewports and Clipping and Rasterization.

Introduction to Procedural Shaders

In Microsoft® Direct3D® for Microsoft DirectX® 8.0, procedural models are used for specifying the behavior of the vertex transformation and lighting pipeline and the pixel texture blending pipeline. There are many advantages to a program model-based syntax for specifying the behavior of the hardware.

First, a procedural model allows a more general syntax for specifying common operations. Fixed-function, as opposed to programmable, APIs must define modes, flags, and so on for an increasing number of operations that need to be expressed. Further, with the increasing power of the hardware—more colors, more textures, more vertex streams, an so on—the token space for the operations multiplied by the data inputs becomes complex. A programmability model, on the other hand, enables even simple operations such as getting the right color and right texture into the right part of the lighting model in a more direct fashion. You do not have to search through all the possible modes; you just have to learn the machine architecture and specify the desired algorithm to be performed.

For example, the following well-known features can be supported.

- Basic geometry transformations
- Simple lighting models
- Vertex blending for skinning
- Vertex morphing (tweening)
- Texture transforms

- Texture generation
- Environment mapping

Second, a procedural model provides an easy mechanism for developing new operations. There are many operations that developers find they need that are not supported in current APIs. In most cases, this is not due to limitations in the capabilities of the hardware but rather to restrictions in the APIs. In general, these operations are also simpler and therefore faster than trying to extract the same behavior by contorting a fixed function API to an extent beyond its designer's expectations.

Examples of new features expected to be commonly implemented include the following:

- *Matrix Palette Skinning*. Character animation with 8-10 bones per mesh.
- *Anisotropic Lighting*. Lighting that currently can be done only at the cost of textures for look-up tables.
- *Membrane Shaders*. Shaders for balloons, skin, and so on (1/cos(eye DOT normal).
- *Kubelka-Munk Shaders*. Shaders that take into account light that penetrates the surface.
- *Procedural Geometry*. Compositing meshes with procedural ones (spheres) to simulate muscles moving under the skin.
- *Displacement Mapping*. Modifying a mesh with a wave pattern or hump that can be tiled/repeated.

Third, a procedural model provides for scalability and evolvability. Hardware capabilities are continuing to evolve rapidly, and programmatic representations can help adapt the API because they scale very well. New features and capabilities can be easily exposed in an incremental way by the following operations.

- Adding new instructions
- Adding new data inputs
- Adding new capabilities from the fixed-function to the programmable portion of the pipeline

Code is the representation that has the best scaling properties for representing complexity. Further, the amount of code that must change inside Direct3D is very small for new features added to the programmable shaders.

Fourth, a procedural model offers familiarity. Software developers understand programming better than they do hardware. An API that truly caters to software developers should map hardware functionality into a code paradigm.

Fifth, a procedural model follows in the footsteps of a photo-real rendering heritage. There has been a tradition of using programmable shaders in high-end photo-real rendering for many years. In general, this area is unconstrained by performance, so

programmable shaders represent the ultimate no-compromise goal for rendering technologies.

Lastly, a procedural model enables direct mapping to the hardware. Most current 3-D hardware, at the vertex processing stage at least, is actually fairly programmable. The programmability through the API enables the application to map directly to this hardware. This enables an you to manage the hardware resources according to their requirements. With a limited set of registers or instructions that can be executed, it is difficult to make a fixed-function implementation that can have all its features enabled independently. If you turn on too many features that require a shared resource, they can stop working in unexpected ways. The programmable API model follows in the DirectX tradition of eliminating this problem by letting the application developer talk directly to the hardware, making any such limitations transparent.

Integration of Vertex Shaders into the Geometry Pipeline

When in operation, a programmable vertex shader replaces the transformation and lighting module in the Microsoft® Direct3D® geometry pipeline. In effect, state information regarding transformation and lighting operations are ignored. However, when the vertex shader is disabled and fixed function processing is returned, all current state settings apply.

Any tessellation of high-order primitives should be done before execution of the vertex shader. Implementations that perform surface tessellation after the shader processing must do so in a way that is not apparent to the application and shader code. Because no semantic information is normally provided before the shader, a special token is used to identify which input stream component represents the base position relative to which all other components are interpolated. No non-interpolable data channels are supported.

On output, the vertex shader must generate vertex positions in homogeneous clip space. Additional data that can be generated includes texture coordinates, colors, fog factors and so on.

The standard graphics pipeline processes the vertices output by the shader, including the following tasks.

- Primitive assembly
- Clipping against the frustum and user clipping planes
- Homogeneous divide
- Viewport scaling
- Backface and viewport culling
- Triangle setup
- Rasterization

Note that the clipping space for DirectX 8.0 vertex shaders is the same as for DirectX 7.0 and DirectX 8.0 fixed function vertex processing. For details, see Clipping Volumes.

Programmable geometry is a mode within the Direct3D application programming interface (API). When it is enabled, it partially replaces the vertex pipeline. When it is disabled, the API has normal control—operating as in DirectX 6.0 and 7.0. Execution of vertex shaders does not change the internal Direct3D state, and no Direct3D state is available for shaders.

[C++]
Calling **IDirect3DDevice8::CreateVertexShader** with the *pFunction* parameter equal to NULL is used to create a shader for the fixed-function pipeline. When *pFunction* is not NULL, the shader is programmable. A call to **IDirect3DDevice8::SetVertexShader** sets the current active shader, which defines whether the rendering pipeline should use programmable or fixed-function vertex processing.

[Visual Basic]
Calling **Direct3DDevice8.CreateVertexShader** with the *FunctionTokenArray* parameter equal to ByVal 0, is used to create a shader for the fixed-function pipeline. When *FunctionTokenArray* is not ByVal 0, the shader is programmable. A call to **Direct3DDevice8.SetVertexShader** sets the current active shader, which defines whether the rendering pipeline should use programmable or fixed-function vertex processing.

Presenting Vertex Data to the Vertex Processor

Vertex data can be presented to the vertex processor using one of the following formats.

- Legacy FVF Format
- Programmable Stream Model

Note

The vertex data format that you choose is independent of and orthogonal to the vertex processing method used (fixed function vs. programmable) For example, you can use streams with fixed function transformation and lighting pipelines and you can use FVF vertices with programmable vertex shaders.

Legacy FVF Format

[C++]
To use legacy FVF formats, use an FVF instead of a handle when calling the **IDirect3DDevice8::SetVertexShader** method as shown in the code example below.

```
g_d3dDevice->SetVertexShader( CUSTOM_FVF );
```

[Visual Basic]
To use legacy FVF formats, use an FVF instead of a handle when calling the **Direct3DDevice8.SetVertexShader** method as shown in the code example below.

```
Call m_D3DDevice.SetVertexShader(CUSTOM_FVF)
```

This following topics cover legacy FVF format types.

- Vertex Legacy Type
- LVertex Legacy Type
- TLVertex Legacy Type

Vertex Legacy Type

This topic shows the steps necessary to initialize and use vertices that have a position, a normal, and texture coordinates.

[C++]
The first step is to define the custom vertex type and FVF as shown in the code example below.

```
struct Vertex
{
    FLOAT x, y, z;
    FLOAT nx, ny, nz;
    FLOAT tu, tv;
};

const DWORD VertexFVF = ( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1 );
```

The next step is to create a vertex buffer with enough room for four vertices by using the **IDirect3DDevice8::CreateVertexBuffer** method as shown in the code example below.

```
g_d3dDevice->CreateVertexBuffer(
            4*sizeof(Vertex), VertexFVF,
            D3DUSAGE_WRITEONLY,
            D3DPOOL_DEFAULT, &pBigSquareVB);
```

The next step is to manipulate the values for each vertex as shown in the code example below.

```
Vertex * v;
pBigSquareVB->Lock( 0, 0, (BYTE**)&v, 0 );
```

```
v[0].x  = 0.0f;  v[0].y  = 10.0;  v[0].z  = 10.0f;
v[0].nx = 0.0f;  v[0].ny = 1.0f;  v[0].nz = 0.0f;
v[0].tu = 0.0f;  v[0].tv = 0.0f;

v[1].x  = 0.0f;  v[1].y  = 0.0f;  v[1].z  = 10.0f;
v[1].nx = 0.0f;  v[1].ny = 1.0f;  v[1].nz = 0.0f;
v[1].tu = 0.0f;  v[1].tv = 0.0f;

v[2].x  = 10.0f; v[2].y  = 10.0f; v[2].z  = 10.0f;
v[2].nx = 0.0f;  v[2].ny = 1.0f;  v[2].nz = 0.0f;
v[2].tu = 0.0f;  v[2].tv = 0.0f;

v[3].x  = 0.0f; v[3].y  = 10.0f;  v[3].z = 10.0f;
v[3].nx = 0.0f; v[3].ny = 1.0f;   v[3].nz = 0.0f;
v[3].tu = 0.0f; v[3].tv = 0.0f;

pBigSquareVB->Unlock();
```

The vertex buffer has been initialized and is ready to render. The following code example shows how to use the legacy FVF to draw a square.

```
g_d3dDevice->SetVertexShader( VertexFVF );
g_d3dDevice->SetStreamSource( 0, pBigSquareVB, 4*sizeof(Vertex) );
g_d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0 ,2);
```

Passing an FVF to the **IDirect3DDevice8::SetVertexShader** method tells Direct3D that a legacy FVF is being used and that stream 0 is the only valid stream.

[Visual Basic]

The first step is to define the custom vertex type and FVF as shown in the code example below.

```
Private Type Vertex
    x As Single
    y As Single
    z As Single
    nx As Single
    ny As Single
    nz As Single
    tu As Single
    tv As Single
End Type

Const VertexFVF = (D3DFVF_XYZ Or D3DFVF_NORMAL Or D3DFVF_TEX1)
```

The next step is to create a vertex buffer with enough room for four vertices by using the **Direct3DDevice8.CreateVertexBuffer** method as shown in the code example below.

```
Set BigSquareVB = m_D3DDevice.CreateVertexBuffer( _
                        4*len(Vertex), VertexFVF, _
                        D3DUSAGE_WRITEONLY, _
                        D3DPOOL_DEFAULT)
```

The next step is to manipulate the values for each vertex as shown in the code example below.

```
Dim v(4) As Vertex
Call BigSquareVB.Lock(0, 0, v(), 0)

v(0).x  = 0.0: v(0).y  = 10.0: v(0).z  = 10.0
v(0).nx = 0.0: v(0).ny = 1.0: v(0).nz = 0.0
v(0).tu = 0.0: v(0).tv = 0.0

v(1).x  = 0.0: v(1).y  = 0.0:  v(1).z  = 10.0
v(1).nx = 0.0: v(1).ny = 1.0:  v(1).nz = 0.0
v(1).tu = 0.0: v(1).tv = 0.0

v(2).x  = 10.0: v(2).y  = 10.0: v(2).z  = 10.0
v(2).nx = 0.0: v(2).ny = 1.0:  v(2).nz = 0.0
v(2).tu = 0.0: v(2).tv = 0.0

v(3).x  = 0.0: v(3).y  = 10.0: v(3).z = 10.0
v(3).nx = 0.0: v(3).ny = 1.0: v(3).nz = 0.0
v(3).tu = 0.0: v(3).tv = 0.0

BigSquareVB.Unlock
```

The vertex buffer has been initialized and is ready to render. The following code example shows how to use the legacy FVF to draw a square.

```
Call m_D3DDevice.SetVertexShader(VertexFVF)
Call m_D3DDevice.SetStreamSource(0, BigSquareVB, 4*len(Vertex))
Call m_D3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0 ,2)
```

Passing an FVF to the **Direct3DDevice8.SetVertexShader** method tells Direct3D that a legacy FVF is being used and that stream 0 is the only valid stream.

LVertex Legacy Type

This topic shows the steps necessary to initialize and use vertices that have a position, diffuse color, specular color, and texture coordinates.

[C++]
The first step is to define the custom vertex type and FVF as shown in the code example below.

```
struct LVertex
{
   FLOAT    x, y, z;
   D3DCOLOR specular, diffuse;
   FLOAT    tu, tv;
};

const DWORD VertexFVF = (D3DFVF_XYZ | D3DFVF_DIFFUSE |
                  D3DFVF_SPECULAR | D3DFVF_TEX1 );
```

The next step is to create a vertex buffer with enough room for four vertices by using the **IDirect3DDevice8::CreateVertexBuffer** method as shown in the code example below.

```
g_d3dDevice->CreateVertexBuffer(
            4*sizeof(LVertex), VertexFVF,
            D3DUSAGE_WRITEONLY,
            D3DPOOL_DEFAULT, &pBigSquareVB);
```

The next step is to manipulate the values for each vertex as shown in the code example below.

```
LVertex * v;
pBigSquareVB->Lock( 0, 0, (BYTE**)&v, 0 );

v[0].x  = 0.0f;  v[0].y  = 10.0;  v[0].z  = 10.0f;
v[0].diffuse  = 0xffff0000;
v[0].specular = 0xff00ff00;
v[0].tu = 0.0f;  v[0].tv = 0.0f;

v[1].x  = 0.0f;  v[1].y  = 0.0f;  v[1].z  = 10.0f;
v[1].diffuse  = 0xff00ff00;
v[1].specular = 0xff00ffff;
v[1].tu = 0.0f;  v[1].tv = 0.0f;

v[2].x  = 10.0f; v[2].y  = 10.0f; v[2].z  = 10.0f;
v[2].diffuse  = 0xffff00ff;
v[2].specular = 0xff000000;
v[2].tu = 0.0f;  v[2].tv = 0.0f;
```

```
v[3].x  = 0.0f; v[3].y  = 10.0f;  v[3].z = 10.0f;
v[3].diffuse  = 0xffffff00;
v[3].specular = 0xffff0000;
v[3].tu = 0.0f; v[3].tv = 0.0f;

pBigSquareVB->Unlock();
```

The vertex buffer has been initialized and is ready to render. The following code example shows how to use the legacy FVF to draw a square.

```
g_d3dDevice->SetVertexShader( VertexFVF );
g_d3dDevice->SetStreamSource( 0, pBigSquareVB, 4*sizeof(LVertex) );
g_d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0 ,2);
```

Passing an FVF to the **IDirect3DDevice8::SetVertexShader** method tells Direct3D that a legacy FVF is being used and that stream 0 is the only valid stream.

[Visual Basic]
The first step is to define the custom vertex type and FVF as shown in the code example below.

```
Private Type LVertex
   x As Single
   y As Single
   z As Single
   diffuse As Long
   specular As Long
   tu As Single
   tv As Single
End Type

Const VertexFVF = (D3DFVF_XYZ Or D3DFVF_DIFFUSE Or _
          D3DFVF_SPECULAR Or D3DFVF_TEX1)
```

The next step is to create a vertex with enough room for four vertices buffer by using the **Direct3DDevice8.CreateVertexBuffer** method as shown in the code example below.

```
Set BigSquareVB = m_D3DDevice.CreateVertexBuffer( _
                        4*len(LVertex), VertexFVF, _
                        D3DUSAGE_WRITEONLY, _
                        D3DPOOL_DEFAULT)
```

The next step is to manipulate the values for each vertex as shown in the code example below.

```
Dim v(4) As LVertex
Call BigSquareVB.Lock(0, 0, v(), 0)

v(0).x  = 0.0: v(0).y  = 10.0: v(0).z  = 10.0
v(0).diffuse = &HFFFF0000;
v(0).specular = &HFF00FF00;
v(0).tu = 0.0: v(0).tv = 0.0

v(1).x  = 0.0: v(1).y  = 0.0:  v(1).z  = 10.0
v(1).diffuse  = &HFF00FF00;
v(1).specular = &HFF00FFFF;
v(1).tu = 0.0: v(1).tv = 0.0

v(2).x  = 10.0: v(2).y  = 10.0: v(2).z  = 10.0
v(2).diffuse  = &HFFFF00FF;
v(2).specular = &HFF000000;
v(2).tu = 0.0: v(2).tv = 0.0

v(3).x  = 0.0: v(3).y  = 10.0: v(3).z = 10.0
v(3).diffuse  = &HFFFFFF00;
v(3).specular = &HFFFF0000;
v(3).tu = 0.0: v(3).tv = 0.0

BigSquareVB.Unlock
```

The vertex buffer has been initialized and is ready to render. The following code example shows how to use the legacy FVF to draw a square.

```
Call m_D3DDevice.SetVertexShader(VertexFVF)
Call m_D3DDevice.SetStreamSource(0, BigSquareVB, 4*len(LVertex))
Call m_D3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0 ,2)
```

Passing an FVF to the **Direct3DDevice8.SetVertexShader** method tells Direct3D that a legacy FVF is being used and that stream 0 is the only valid stream.

TLVertex Legacy Type

The topic shows the steps necessary to initialize and use vertices that have a transformed position, diffuse color, specular color, and texture coordinates.

[C++]
The first step is to define the custom vertex type and FVF as shown in the code example below.

```
struct TLVertex
{
```

```
    FLOAT    x, y, z, rhw;
    D3DCOLOR specular, diffuse;
    FLOAT    tu, tv;
};

const DWORD VertexFVF = (D3DFVF_XYZRHW | D3DFVF_DIFFUSE |
                         D3DFVF_SPECULAR | D3DFVF_TEX1 );
```

The next step is to create a vertex buffer with enough room for four vertices by using the **IDirect3DDevice8::CreateVertexBuffer** method as shown in the code example below.

```
g_d3dDevice->CreateVertexBuffer(
                4*sizeof(TLVertex), VertexFVF,
                D3DUSAGE_WRITEONLY,
                D3DPOOL_DEFAULT, &pBigSquareVB);
```

The next step is to manipulate the values for each vertex as shown in the code example below.

```
TLVertex * v;
pBigSquareVB->Lock( 0, 0, (BYTE**)&v, 0 );

v[0].x  = 0.0f;  v[0].y  = 10.0;  v[0].z  = 10.0f; v[0].rhw = 1.0f;
v[0].diffuse  = 0xffff0000;
v[0].specular = 0xff00ff00;
v[0].tu = 0.0f;  v[0].tv = 0.0f;

v[1].x  = 0.0f;  v[1].y  = 0.0f;  v[1].z  = 10.0f; v[1].rhw = 1.0f;
v[1].diffuse  = 0xff00ff00;
v[1].specular = 0xff00ffff;
v[1].tu = 0.0f;  v[1].tv = 0.0f;

v[2].x  = 10.0f; v[2].y  = 10.0f; v[2].z  = 10.0f; v[2].rhw = 1.0f;
v[2].diffuse  = 0xffff00ff;
v[2].specular = 0xff000000;
v[2].tu = 0.0f;  v[2].tv = 0.0f;

v[3].x  = 0.0f; v[3].y  = 10.0f;  v[3].z = 10.0f; v[3].rhw = 1.0f;
v[3].diffuse  = 0xffffff00;
v[3].specular = 0xffff0000;
v[3].tu = 0.0f; v[3].tv = 0.0f;

pBigSquareVB->Unlock();
```

The vertex buffer has been initialized and is ready to render. The following code example shows how to use the legacy FVF to draw a square.

```
g_d3dDevice->SetVertexShader( VertexFVF );
g_d3dDevice->SetStreamSource( 0, pBigSquareVB, 4*sizeof(TLVertex) );
g_d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0 ,2);
```

Passing an FVF to the **IDirect3DDevice8::SetVertexShader** method tells Direct3D that a legacy FVF is being used and that stream 0 is the only valid stream.

[Visual Basic]
The first step is to define the custom vertex type and FVF as shown in the code example below.

```
Private Type TLVertex
   x As Single
   y As Single
   z As Single
   rhw As Single
   diffuse As Long
   specular As Long
   tu As Single
   tv As Single
End Type

Const VertexFVF = (D3DFVF_XYZRHW Or D3DFVF_DIFFUSE Or _
          D3DFVF_SPECULAR Or D3DFVF_TEX1)
```

The next step is to create a vertex buffer with enough room for four vertices by using the **Direct3DDevice8.CreateVertexBuffer** method as shown in the code example below.

```
Set BigSquareVB = m_D3DDevice.CreateVertexBuffer( _
                    4*len(TLVertex), VertexFVF, _
                    D3DUSAGE_WRITEONLY, D3DPOOL_DEFAULT)
```

The next step is to manipulate the values for each vertex as shown in the code example below.

```
Dim v(4) As TLVertex
Call BigSquareVB.Lock(0, 0, v(), 0)

v(0).x  = 0.0: v(0).y  = 10.0: v(0).z  = 10.0: v(0).rhw = 1.0
v(0).diffuse = &HFFFF0000;
v(0).specular = &HFF00FF00;
v(0).tu = 0.0: v(0).tv = 0.0

v(1).x  = 0.0: v(1).y  = 0.0:  v(1).z  = 10.0: v(1).rhw = 1.0
v(1).diffuse  = &HFF00FF00;
```

```
v(1).specular = &HFF00FFFF;
v(1).tu = 0.0: v(1).tv = 0.0

v(2).x  = 10.0: v(2).y  = 10.0: v(2).z  = 10.0: v(2).rhw = 1.0
v(2).diffuse  = &HFFFF00FF;
v(2).specular = &HFF000000;
v(2).tu = 0.0: v(2).tv = 0.0

v(3).x  = 0.0: v(3).y  = 10.0: v(3).z = 10.0: v(3).rhw = 1.0
v(3).diffuse  = &HFFFFFF00;
v(3).specular = &HFFFF0000;
v(3).tu = 0.0: v(3).tv = 0.0

BigSquareVB.Unlock
```

The vertex buffer has been initialized and is ready to render. The following code example shows how to use the legacy FVF to draw a square.

```
Call m_D3DDevice.SetVertexShader(VertexFVF)
Call m_D3DDevice.SetStreamSource(0, BigSquareVB, 4*len(TLVertex))
Call m_D3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0 ,2)
```

Passing an FVF to the **Direct3DDevice8.SetVertexShader** method tells Direct3D that a legacy FVF is being used and that stream 0 is the only valid stream.

Programmable Stream Model

This following sections cover shaders that can be used for the programmable stream model.

- ColorVertex Shader
- SingleTexture Shader
- MultiTexture Shader

ColorVertex Shader

This topic shows the steps necessary to initialize and use a simple vertex shader that uses a position and a diffuse color.

[C++]
The first step is to declare the structures that hold the position and color as shown in the code example below.

```
struct XYZBuff
{
   FLOAT x, y, z;
```

```
    D3DCOLOR color;
};

struct ColBuf
{
    D3DCOLOR color;
};

#define XYZBUFF (D3DFVF_XYZ)
#define COLBUFF (D3DFVF_DIFFUSE)
```

The next step is to create a Vertex Shader Declaration as shown in the code below.

```
DWORD decl[] =
{
    D3DVSD_STREAM(0),
    D3DVSD_REG( D3DVSDE_POSITION, D3DVSDT_FLOAT3 ),
    D3DVSD_STREAM(1),
    D3DVSD_REG( D3DVSDE_DIFFUSE, D3DVSDT_D3DCOLOR),
    D3DVSD_END()
};
```

The next step is to call the **IDirect3DDevice8::CreateVertexShader** method to create the vertex shader.

```
g_d3dDevice->CreateVertexShader( decl, NULL, &vShader, 0);
```

Passing NULL to the second parameter of **CreateVertexShader** tells Direct3D that this vertex shader will use a fixed function pipeline.

After creating the vertex buffer and vertex shader, they are ready to use. The code example below shows how to set the vertex shader, set the stream source, and then draw a triangle list that uses the new vertex shader.

```
g_d3dDevice->SetVertexShader( &vShader );
g_d3dDevice->SetStreamSource( 0, xyzbuf, 4 * sizeof(FLOAT));
g_d3dDevice->SetStreamSource( 1, colbuf, 2 * sizeof(FLOAT));
g_d3dDevice->SetIndices( pIB, 0 );
g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0, count / 3 );
```

[Visual Basic]

The first step is to declare the structures that hold the position and color as shown in the code example below.

```
Private Type XYZBuff
    x As Single
    y As Single
    z As Single
```

```
End Type

Private Type ColBuf
   color As Long
End Type

Const XYZBUFF_FVF = (D3DFVF_XYZ)
Const COLBUFF_FVF = (D3DFVF_DIFFUSE)
```

The next step is to create a Vertex Shader Declaration as shown in the code below.

```
Dim decl(5) As Long

decl(0) = D3DVSD_STREAM(0)
decl(1) = D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3)
decl(2) = D3DVSD_STREAM(1)
decl(3) = D3DVSD_REG( D3DVSDE_DIFFUSE, D3DVSDT_UBYTE)
decl(4) = D3DVSD_END()
```

The next step is to call the **Direct3DDevice8.CreateVertexShader** method to create the vertex shader.

```
Call m_D3DDevice.CreateVertexShader( decl, ByVal 0, vShader, 0)
```

Passing ByVal 0 to the second parameter of **CreateVertexShader** tells Direct3D that this vertex shader will use a fixed function pipeline.

After creating the vertex buffer and vertex shader, they are ready to use. The code example below shows how to set the vertex shader, set the stream source, and then draw a triangle list that uses the new vertex shader.

```
Call m_D3DDevice.SetVertexShader( vShader )
Call m_D3DDevice.SetStreamSource( 0, xyzbuf, 4 * len(xyzbuf))
Call m_D3DDevice.SetStreamSource( 1, colbuf, 2 * len(colbuf))
Call m_D3DDevice.SetIndices( IB, 0 )
Call m_D3DDevice.DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0,
count / 3 )
```

SingleTexture Shader

This topic shows the steps necessary to initialize and use a simple vertex shader that uses a position and texture coordinates.

[C++]
The first step is to declare the structures that hold the position and texture coordinates as shown in the code example below.

```
struct XYZBuff
{
   D3DVALUE x, y, z;
};

struct TEX0Buff
{
   D3DVALUE tu, tv;
};

#define XYZBUFF  (D3DFVF_XYZ)
#define TEX0BUFF (D3DFVF_TEX1)
```

The next step is to create a Vertex Shader Declaration as shown in the code below.

```
DWORD decl[] =
{
   D3DVSD_STREAM(0),
   D3DVSD_REG( D3DVSDE_POSITION, D3DVSDT_FLOAT3 ),
   D3DVSD_STREAM(1),
   D3DVSD_REG( D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2 ),
   D3DVSD_END()
};
```

The next step is to call the **IDirect3DDevice8::CreateVertexShader** method to create the vertex shader.

```
g_d3dDevice->CreateVertexShader( decl, NULL, &vShader, 0);
```

Passing NULL to the second parameter of **CreateVertexShader** tells Direct3D that this vertex shader will use a fixed function pipeline.

After creating the vertex buffer and vertex shader, they are ready to use. The code example below shows how to set the vertex shader, set the stream source, and then draw a triangle list that uses the new vertex shader.

```
g_d3dDevice->SetVertexShader( &vShader );
g_d3dDevice->SetStreamSource( 0, xyzbuf, 4 * sizeof(FLOAT));
g_d3dDevice->SetStreamSource( 1, tex0buf, 2 * sizeof(FLOAT));
g_d3dDevice->SetIndices( pIB, 0 );
g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0, count / 3 );
```

[Visual Basic]

The first step is to declare the structures that hold the position and color as shown in the code example below.

```
Private Type XYZBuff
   x As Single
```

```
    y As Single
    z As Single
End Type

Private Type Tex0Buf
    tu As Single
    tv As Single
End Type

Const XYZBUFF_FVF = (D3DFVF_XYZ)
Const TEX0BUFF_FVF = (D3DFVF_TEX1)
```

The next step is to create a Vertex Shader Declaration as shown in the code below.

```
Dim decl(5) As Long

decl(0) = D3DVSD_STREAM(0)
decl(1) = D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3)
decl(2) = D3DVSD_STREAM(1)
decl(3) = D3DVSD_REG(D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2)
decl(4) = D3DVSD_END()
```

The next step is to call the **Direct3DDevice8.CreateVertexShader** method to create the vertex shader.

```
Call m_D3DDevice.CreateVertexShader( decl, ByVal 0, vShader, 0)
```

Passing ByVal 0 to the second parameter of **CreateVertexShader** tells Direct3D that this vertex shader will use a fixed function pipeline.

After creating the vertex buffer and vertex shader, they are ready to use. The code example below shows how to set the vertex shader, set the stream source, and then draw a triangle list that uses the new vertex shader.

```
Call m_D3DDevice.SetVertexShader( vShader )
Call m_D3DDevice.SetStreamSource( 0, xyzbuf, 4 * len(xyzbuf))
Call m_D3DDevice.SetStreamSource( 1, tex0buf, 2 * len(colbuf))
Call m_D3DDevice.SetIndices( IB, 0 )
Call m_D3DDevice.DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0,
count / 3 )
```

MultiTexture Shader

This topic shows the steps necessary to initialize and use a simple vertex shader that uses a position and multiple texture coordinates for multiple textures.

[C++]

The first step is to declare the structures that holds the position and color as shown in the code example below.

```
struct XYZBuff
{
   D3DVALUE x, y, z;
};

struct Tex0Buff
{
   D3DVALUE tu, tv;
};

struct Tex1Buff
{
   D3DVALUE tu2, tv2;
};

#define XYZBUFF (D3DFVF_XYZ)
#define TEX0BUFF (D3DFVF_TEX1)
#define TEX1BUFF (D3DFVF_TEX1)
```

The next step is to create a Vertex Shader Declaration as shown in the code below.

```
DWORD decl[] =
{
   D3DVSD_STREAM(0),
   D3DVSD_REG( D3DVSDE_POSITION, D3DVSDT_FLOAT3 ),
   D3DVSD_STREAM(1),
   D3DVSD_REG( D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2),
   D3DVSD_STREAM(2),
   D3DVSD_REG( D3DVSDE_TEXCOORD1, D3DVSDT_FLOAT2),
   D3DVSD_END()
};
```

The next step is to call the **IDirect3DDevice8::CreateVertexShader** method to create the vertex shader.

```
g_d3dDevice->CreateVertexShader( decl, NULL, &vShader, 0);
```

Passing NULL to the second parameter of **CreateVertexShader** tells Direct3D that this vertex shader will use a fixed function pipeline.

After creating the vertex buffer and vertex shader, they are ready to use. The code example below shows how to set the vertex shader, set the stream source, and then draw a triangle list that uses the new vertex shader.

```
g_d3dDevice->SetVertexShader( &vShader );
g_d3dDevice->SetStreamSource( 0, xyzbuf, 4 * sizeof(FLOAT));
```

```
g_d3dDevice->SetStreamSource( 1, tex0buf, 2 * sizeof(FLOAT));
g_d3dDevice->SetStreamSource( 2, tex1buf, 2 * sizeof(FLOAT));
g_d3dDevice->SetIndices( pIB, 0 );
g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0, count / 3 );
```

[Visual Basic]
The first step is to declare the structures that holds the position and color as shown in the code example below.

```
Private Type XYZBUFF
   x As Single
   y As Single
   z As Single
End Type

Private Type TEX0BUFF
   tu As Single
   tv As Single
End Type

Private Type TEX1BUFF
   tu2 As Single
   tv2 As Single
End Type

Const XYZBUFF_FVF = (D3DFVF_XYZ)
Const TEX0BUFF_FVF = (D3DFVF_TEX1)
Const TEX1BUFF_FVF = (D3DFVF_TEX1)
```

The next step is to create a Vertex Shader Declaration as shown in the code below.

```
Dim decl(7) As Long

decl(0) = D3DVSD_STREAM(0)
decl(1) = D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3)
decl(2) = D3DVSD_STREAM(1)
decl(3) = D3DVSD_REG( D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2)
decl(4) = D3DVSD_STREAM(2)
decl(5) = D3DVSD_REG( D3DVSDE_TEXCOORD1, D3DVSDT_FLOAT2)
decl(6) = D3DVSD_END()
```

The next step is to call the **Direct3DDevice8.CreateVertexShader** method to create the vertex shader.

```
Call m_D3DDevice.CreateVertexShader( decl, ByVal 0, vShader, 0)
```

Passing ByVal 0 to the second parameter of **CreateVertexShader** tells Direct3D that this vertex shader will use a fixed function pipeline.

After creating the vertex buffer and vertex shader, they are ready to use. The code example below shows how to set the vertex shader, set the stream source, and then draw a triangle list that uses the new vertex shader.

```
Call m_D3DDevice.SetVertexShader( vShader )
Call m_D3DDevice.SetStreamSource( 0, xyzbuf, 4 * len(xyzbuf))
Call m_D3DDevice.SetStreamSource( 1, tex0buf, 2 * len(colbuf))
Call m_D3DDevice.SetStreamSource( 2, tex0buf, 2 * len(colbuf))
Call m_D3DDevice.SetIndices( IB, 0 )
Call m_D3DDevice.DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, max - min + 1, 0,
count / 3 )
```

Using DirectX Graphics

This section is a guide to using the Microsoft® Direct3D® and Direct3DX APIs in application development.
Information is presented in the following topics.

- Direct3D Object
- Direct3D Device
- Direct3D Resources
- Surfaces
- Lights
- Materials
- Vertex Formats
- Textures
- Depth Buffers
- Stencil Buffers
- Vertex Buffers
- Index Buffers
- Techniques and Special Effects

For a more general overview, see Getting Started with DirectX Graphics.
For an understanding of the general mechanisms of data flow and organization, see Understanding DirectX Graphics.
For information on advanced features such as programmable shaders, see Advanced Topics in DirectX Graphics.

Direct3D Object

This section contains information about the Microsoft® Direct3D® object.

- What is a Direct3D Object?
- Accessing Direct3D

What is a Direct3D Object?

Microsoft® Direct3D® is implemented through COM objects and interfaces. Applications written in C++ access these interfaces and objects directly, whereas Microsoft Visual Basic® applications interact with a layer of code—visible as the Microsoft DirectX® for Visual Basic Classes—that marshals data from a Visual Basic application to the DirectX run time.

The Direct3D object is the first object that your application creates and the last object that your application releases. Functions for enumerating and retrieving capabilities of a Direct3D device are accessible through the Direct3D object. This enables applications to select devices without creating them.

Accessing Direct3D

[C++]

When a Microsoft® Direct3D® application written in C++ starts, it must obtain a pointer to an **IDirect3D8** interface to access Direct3D functionality.

The following code example shows how to use the **Direct3DCreate8** function to retrieve a pointer to the Direct3D interface.

```
LPDIRECT3D8 g_pD3D = NULL;

    if( NULL == (g_pD3D = Direct3DCreate8(D3D_SDK_VERSION)))
        return E_FAIL;
```

To navigate from the Direct3DDevice object to the Direct3D object that created the device, use the **IDirect3DDevice8::GetDirect3D** method.

[Visual Basic]

When a Microsoft® Direct3D® application written in Microsoft Visual Basic® starts, it must obtain a reference to the **Direct3D8** class to access Direct3D functionality.

The following code example shows how to use the **DirectX8.Direct3DCreate** method to retrieve a reference to the Direct3D class.

```
Dim g_DX As New DirectX8
Dim g_D3D As Direct3D8

    Set g_D3D = g_DX.Direct3DCreate()

    If g_D3D Is Nothing Then Exit Function
```

To navigate from the Direct3DDevice object to the Direct3D object that created the device, use the **Direct3DDevice8.GetDirect3D** method.

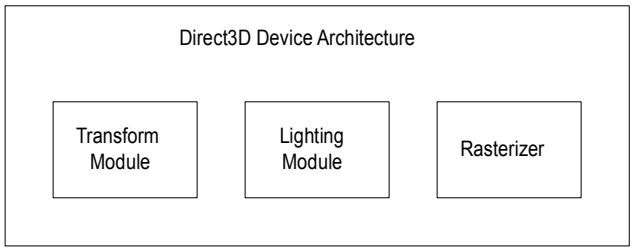
Direct3D Device

This section provides an overview of Microsoft® Direct3D® devices. The overview is divided into the following topics.

- What is a Direct3D Device?
- Device Types
- Device-Supported Primitive Types
- Using Devices
- Device States
- Lost Devices

What is a Direct3D Device?

A Microsoft® Direct3D® device is the rendering component of Direct3D. It encapsulates and stores the rendering state. In addition, a Direct3D device performs transformations and lighting operations and rasterizes an image to a surface. Architecturally, Direct3D devices contain a transformation module, a lighting module, and a rasterizing module, as the following illustration shows.



Direct3D enables applications that use custom transformation and lighting models to bypass the Direct3D device's transformation and lighting modules. For details, see Vertex Shaders.

Device Types

This section introduces Microsoft® Direct3D® devices and presents information for each type of device. The following topics are discussed.

- About Device Types
- HAL Device

- Reference Device
- Pluggable Software Device
- Device Behaviors

About Device Types

Microsoft® Direct3D® currently supports three main types of Direct3D devices: a hardware abstraction layer (HAL) device with hardware-accelerated rasterization and shading with both hardware and software vertex processing; a reference device; and a pluggable software device (providing software rasterization).
You can think of these three devices as three separate drivers. Software and reference devices are represented by software drivers, and the HAL device is represented by a hardware driver. The most common way to take advantage of these devices is to use the HAL device for shipping applications, and the reference device for feature testing. These are provided by third parties to emulate particular devices—for example, developmental hardware that has not yet been released.
The Direct3D device that an application creates must correspond to the capabilities of the hardware on which the application is running. Direct3D provides rendering capabilities, either by accessing 3-D hardware that is installed in the computer or by emulating the capabilities of 3-D hardware in software. Therefore, Direct3D provides devices for both hardware access and software emulation.
Hardware-accelerated devices give much better performance than software devices. The HAL device type is available on all Direct3D supported graphic adapters. In most cases, applications target computers that have hardware acceleration and rely on software emulation to accommodate lower-end computers.
With the exception of the reference device, software devices do not always support the same features as a hardware device. Applications should always query for device capabilities to determine which features are supported.
Because the behavior of the software and reference devices provided with Microsoft DirectX® 8.0 is identical to that of the HAL device, application code authored to work with the HAL device will work with the software or reference devices without modifications. Note that while the provided software or reference device behavior is identical to that of the HAL device, the device capabilities do vary, and a particular software device may implement a much smaller set of capabilities.

HAL Device

The primary device type is the hardware abstraction layer (HAL) device, which supports hardware accelerated rasterization and both hardware and software vertex processing. If the computer on which your application is running is equipped with a display adapter that supports Microsoft® Direct3D®, your application should use it for 3-D operations. Direct3D HAL devices implement all or part of the transformation, lighting, and rasterizing modules in hardware.
Applications do not access 3-D cards directly. They call Direct3D functions and methods. Direct3D accesses the hardware through the HAL. If the computer that your application is running on supports the HAL, it will gain the best performance by using a HAL device.

[C++]
To create a HAL device from C++, call the **IDirect3D8::CreateDevice** method, and pass the D3DDEVTYPE_HAL constant as the device type. For details, see Creating a Device.

[Visual Basic]
To create a HAL device from Microsoft Visual Basic®, call the **Direct3D8.CreateDevice** method, and pass D3DDEVTYPE_HAL constant as the device type. For details, see Creating a Device.

Note
Hardware devices cannot render to 8-bit render-target surfaces.

Reference Device

Microsoft® Direct3D® supports an additional device type called a reference device or reference rasterizer. Unlike a software device, the reference rasterizer supports every Direct3D feature. Because these features are implemented for accuracy, rather than speed, and are implemented in software, the results are not very fast. The reference rasterizer does make use of special CPU instructions whenever it can, but it is not intended for retail applications. Use the reference rasterizer only for feature testing or demonstration purposes.

[C++]
To create a reference device from C++, call the **IDirect3D8::CreateDevice** method, and pass the D3DDEVTYPE_REF constant as the device type. For details, see Creating a Device.

[Visual Basic]
To create a reference device from Microsoft Visual Basic®, call the **Direct3D8.CreateDevice** method, and pass the D3DDEVTYPE_HAL constant as the device type. For details, see Creating a Device.

Pluggable Software Device

If the user's computer provides no special hardware acceleration for 3-D operations, your application might emulate 3-D hardware in software. Software rasterization devices emulate the functions of color 3-D hardware in software. Because a software device is emulated in software, it runs more slowly than a hardware abstraction layer (HAL) device. However, software devices take advantage of any special instructions supported by the user's CPU to increase performance. Instruction sets include the AMD 3D-Now! instruction set on some AMD processors and the MMX instruction set supported by many Intel processors. Microsoft® Direct3D uses the 3D-Now!

instruction set to accelerate transformation and lighting operations and the MMX instruction set to accelerate rasterization.
Software rasterization for Direct3D® is provided by pluggable software devices, which enable applications to access a variety of software rasterizers through the Direct3D interfaces. Software devices are loaded by the application and registered with the Direct3D object, at which point you can create a Direct3DDevice object that will perform rendering with the software device.
Direct3D software devices communicate with Direct3D through an interface similar to the hardware device driver interface (DDI).
The Direct3D DDK provides the documentation and headers for developing pluggable software devices.

Device Behaviors

[C++]
Microsoft® Direct3D® enables you to specify the behavior of a device, as well the device's type. The **IDirect3D8::CreateDevice** method enables a combination of one or more of the behavior flags to control the global behaviors of the Direct3D device. These behaviors specify what is and is not maintained in the run-time portion of Direct3D, and the device types specify which driver to use. Although some combinations of device behaviors are not valid, it is possible to use all device behaviors with all device types. For example, it is valid to specify D3DDEVTYPE_SW on a device created with D3DCREATE_PUREDEVICE.

[Visual Basic]
Microsoft® Direct3D® enables you to specify the behavior and the type of a device. The **Direct3D8.CreateDevice** method enables a combination of one or more of the behavior flags to control the global behaviors of the Direct3D device. These behaviors specify what is and is not maintained in the run-time portion of Direct3D, and the device types specify which driver to use. Although some combinations of device behaviors are not valid, it is possible to use all device behaviors with all device types. For example, it is valid to specify the D3DDEVTYPE_SW member of the **CONST_D3DDEVTYPE** enumeration on a device created with the D3DCREATE_PUREDEVICE member of the **CONST_D3DCREATEFLAGS** enumeration.

Device-Supported Primitive Types

Microsoft® Direct3D® devices can create and manipulate the following types of primitives.

- Point Lists
- Line Lists
- Line Strips
- Triangle Lists

- Triangle Strips
- Triangle Fans

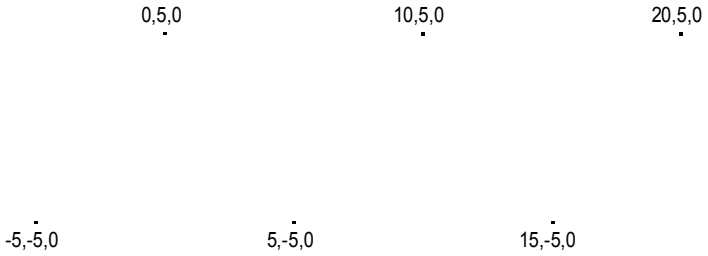
[C++]
You can render primitive types from a C++ application with any of the rendering methods of the **IDirect3DDevice8** interface. For more information, see Rendering.

[Visual Basic]
You can render primitive types from a Microsoft Visual Basic® application with any of the rendering methods of the **Direct3DDevice8** class. For more information, see Rendering.

Note that you cannot render point lists with the indexed-primitive rendering methods.

Point Lists

A point list is a collection of vertices that are rendered as isolated points. Your application can use them in 3-D scenes for star fields, or dotted lines on the surface of a polygon.
The following illustration depicts a rendered point list.



Your application can apply materials and textures to a point list. The colors in the material or texture appear only at the points drawn, and not anywhere between the points.

[C++]
The following code shows how to create vertices for this point list.

```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
```

```
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}
};
```

The code example below shows how to use **IDirect3DDevice8::DrawPrimitive** to render this point list.

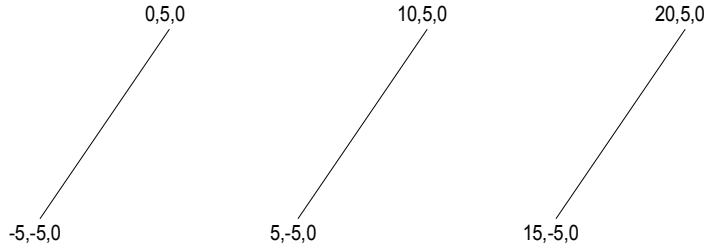
```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_POINTLIST, 0, 6 );
```

[Visual Basic]
The following code shows how to create vertices for this point list.

```
Private Type CUSTOMVERTEX
    x As Single
    y As Single
    z As Single
End Type

Dim Vertices(5) As CUSTOMVERTEX
With Vertices(0): .x = -5.0: .y = -5.0: .z = 0.0: End With
With Vertices(1): .x =  0.0: .y =  5.0: .z = 0.0: End With
With Vertices(2): .x =  5.0: .y = -5.0: .z = 0.0: End With
With Vertices(3): .x = 10.0: .y =  5.0: .z = 0.0: End With
With Vertices(4): .x = 15.0: .y = -5.0: .z = 0.0: End With
With Vertices(5): .x = 20.0: .y =  5.0: .z = 0.0: End With
```

Line Lists

A line list is a list of isolated, straight line segments. Line lists are useful for such tasks as adding sleet or heavy rain to a 3-D scene. Applications create a line list by filling an array of vertices, and the number of vertices in a line list must be an even number greater than or equal to two.
The following illustration shows a rendered line list.



You can apply materials and textures to a line list. The colors in the material or texture appear only along the lines drawn, not at any point in between the lines.

[C++]

The following code shows how to create vertices for this line list.

```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}
};
```

The code example below shows how to use **IDirect3DDevice8::DrawPrimitive** to render this line list.

```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_LINELIST, 0, 3 );
```

[Visual Basic]

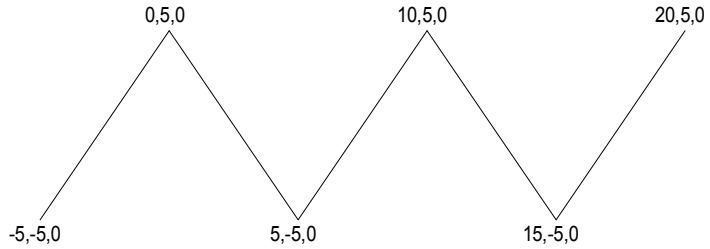
The following code shows how to create vertices for this line list.

```
Private Type CUSTOMVERTEX
    x As Single
    y As Single
    z As Single
End Type

Dim Vertices(5) As CUSTOMVERTEX
```

```
With Vertices(0): .x = -5.0: .y = -5.0: .z = 0.0: End With
With Vertices(1): .x =  0.0: .y =  5.0: .z = 0.0: End With
With Vertices(2): .x =  5.0: .y = -5.0: .z = 0.0: End With
With Vertices(3): .x = 10.0: .y =  5.0: .z = 0.0: End With
With Vertices(4): .x = 15.0: .y = -5.0: .z = 0.0: End With
With Vertices(5): .x = 20.0: .y =  5.0: .z = 0.0: End With
```

Line Strips

A line strip is a primitive that is composed of connected line segments. Your application can use line strips for creating polygons that are not closed. A closed polygon is a polygon whose last vertex is connected to its first vertex by a line segment. If your application makes polygons based on line strips, the vertices are not guaranteed to be coplanar.
The following illustration depicts a rendered line strip.



[C++]
The following code shows how to create vertices for this line strip.

```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}
};
```

The code example below shows how to use **IDirect3DDevice8::DrawPrimitive** to render this line strip.

```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_LINESTRIP, 0, 5 );
```

[Visual Basic]

The following code shows how to create vertices for this line strip.

```
Private Type CUSTOMVERTEX
    x As Single
    y As Single
    z As Single
End Type

Dim Vertices(5) As CUSTOMVERTEX
With Vertices(0): .x = -5.0: .y = -5.0: .z = 0.0: End With
With Vertices(1): .x =  0.0: .y =  5.0: .z = 0.0: End With
With Vertices(2): .x =  5.0: .y = -5.0: .z = 0.0: End With
With Vertices(3): .x = 10.0: .y =  5.0: .z = 0.0: End With
With Vertices(4): .x = 15.0: .y = -5.0: .z = 0.0: End With
With Vertices(5): .x = 20.0: .y =  5.0: .z = 0.0: End With
```

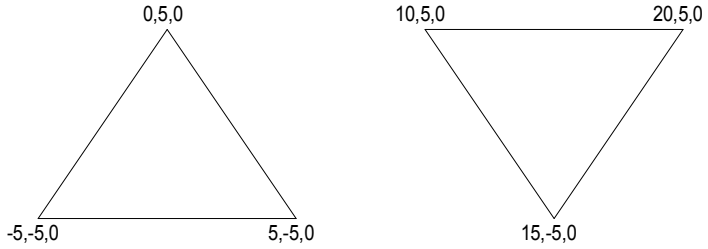
Triangle Lists

A triangle list is a list of isolated triangles. They might or might not be near each other. A triangle list must have at least three vertices. The total number of vertices must be divisible by three.

Use triangle lists to create an object that is composed of disjoint pieces. For instance, one way to create a force-field wall in a 3-D game is to specify a large list of small, unconnected triangles. Then apply a material and texture that appears to emit light to the triangle list. Each triangle in the wall appears to glow. The scene behind the wall becomes partially visible through the gaps between the triangles, as a player might expect when looking at a force field.

Triangle lists are also useful for creating primitives that have sharp edges and are shaded with Gouraud shading. See Face and Vertex Normal Vectors.

The following illustration depicts a rendered triangle list.



[C++]

The following code shows how to create vertices for this triangle list.

```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}

};
```

The code example below shows how to use **IDirect3DDevice8::DrawPrimitive** to render this triangle list.

```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 2 );
```

[Visual Basic]

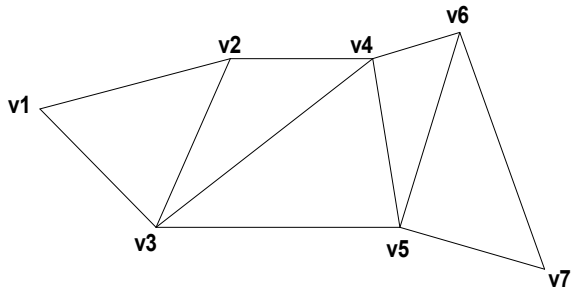
The following code shows how to create vertices for this triangle list.

```
Private Type CUSTOMVERTEX
    x As Single
    y As Single
    z As Single
End Type

Dim Vertices(5) As CUSTOMVERTEX
```

```
With Vertices(0): .x = -5.0: .y = -5.0: .z = 0.0: End With
With Vertices(1): .x =  0.0: .y =  5.0: .z = 0.0: End With
With Vertices(2): .x =  5.0: .y = -5.0: .z = 0.0: End With
With Vertices(3): .x = 10.0: .y =  5.0: .z = 0.0: End With
With Vertices(4): .x = 15.0: .y = -5.0: .z = 0.0: End With
With Vertices(5): .x = 20.0: .y =  5.0: .z = 0.0: End With
```

Triangle Strips

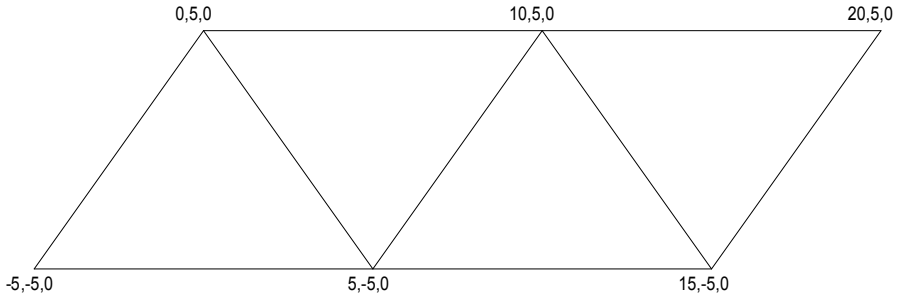
A triangle strip is a series of connected triangles. Because the triangles are connected, the application does not need to repeatedly specify all three vertices for each triangle. For example, you need only seven vertices to define the following triangle strip.



The system uses vertices v1, v2, and v3 to draw the first triangle, v2, v4, and v3 to draw the second triangle, v3, v4, and v5 to draw the third, v4, v6, and v5 to draw the fourth, and so on. Notice that the vertices of the second and fourth triangles are out of order; this is required to make sure that all the triangles are drawn in a clockwise orientation.

Most objects in 3-D scenes are composed of triangle strips. This is because triangle strips can be used to specify complex objects in a way that makes efficient use of memory and processing time.

The following illustration depicts a rendered triangle strip.



[C++]
The following code shows how to create vertices for this triangle strip.

```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    {-5.0, -5.0, 0.0},
    { 0.0,  5.0, 0.0},
    { 5.0, -5.0, 0.0},
    {10.0,  5.0, 0.0},
    {15.0, -5.0, 0.0},
    {20.0,  5.0, 0.0}
};
```

The code example below shows how to use **IDirect3DDevice8::DrawPrimitive** to render this triangle strip.

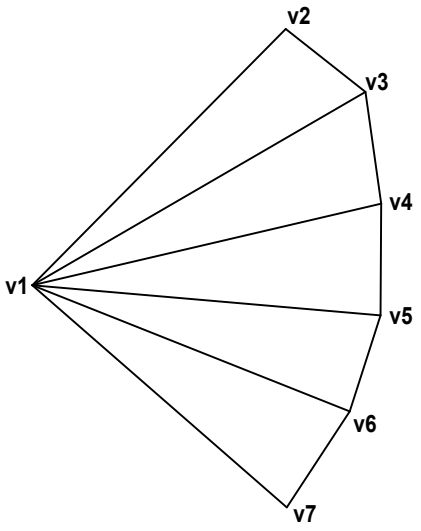
```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 5 );
```

[Visual Basic]
The following code shows how to create vertices for this triangle strip.

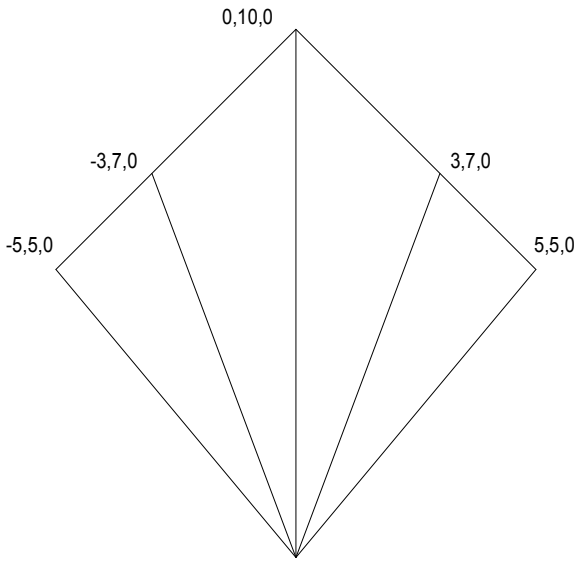
```
Private Type CUSTOMVERTEX
    x As Single
    y As Single
    z As Single
End Type

Dim Vertices(5) As CUSTOMVERTEX
With Vertices(0): .x = -5.0: .y = -5.0: .z = 0.0: End With
With Vertices(1): .x =  0.0: .y =  5.0: .z = 0.0: End With
With Vertices(2): .x =  5.0: .y = -5.0: .z = 0.0: End With
With Vertices(3): .x = 10.0: .y =  5.0: .z = 0.0: End With
With Vertices(4): .x = 15.0: .y = -5.0: .z = 0.0: End With
With Vertices(5): .x = 20.0: .y =  5.0: .z = 0.0: End With
```

Triangle Fans

A triangle fan is similar to a triangle strip, except that all the triangles share one vertex, as shown in the following illustration.



The system uses vertices v2, v3, and v1 to draw the first triangle, v3, v4, and v1 to draw the second triangle, v4, v5, and v1 to draw the third triangle, and so on. When flat shading is enabled, the system shades the triangle with the color from its first vertex.
This illustration depicts a rendered triangle fan.



[C++]

The following code shows how to create vertices for this triangle fan.

```
struct CUSTOMVERTEX
{
    float x,y,z;
};

CUSTOMVERTEX Vertices[] =
{
    { 0.0, 0.0, 0.0},
    {-5.0, -5.0, 0.0},
    {-3.0,  7.0, 0.0},
    { 0.0, 10.0, 0.0},
    { 3.0,  7.0, 0.0},
    { 5.0,  5.0, 0.0},
};
```

The code example below shows how to use **IDirect3DDevice8::DrawPrimitive** to render this triangle fan.

```
//
// It is assumed that d3dDevice is a valid
// pointer to a IDirect3DDevice8 interface.
//
d3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 4 );
```

[Visual Basic]

The following code shows how to create vertices for this triangle fan.

```
Private Type CUSTOMVERTEX
   x As Single
   y As Single
   z As Single
End Type

Dim Vertices(5) As CUSTOMVERTEX
With Vertices(0): .x =  0.0: .y =  0.0: .z = 0.0: End With
With Vertices(1): .x = -5.0: .y = -5.0: .z = 0.0: End With
With Vertices(2): .x = -3.0: .y =  7.0: .z = 0.0: End With
With Vertices(3): .x =  0.0: .y = 10.0: .z = 0.0: End With
With Vertices(4): .x =  3.0: .y =  7.0: .z = 0.0: End With
With Vertices(5): .x =  5.0: .y =  5.0: .z = 0.0: End With
```

Using Devices

This section provides information about using Microsoft® Direct3D® devices in an application. Information is divided into the following topics.

- Determining Hardware Support
- Selecting a Device
- Creating a Device
- Setting Transformations
- Processing Vertex Data
- Working with Mouse Cursors
- Rendering

Determining Hardware Support

Direct3D provides the following functions to determine hardware support.

[C++]

IDirect3D8::CheckDeviceFormat

Used to verify whether a surface format can be used as a texture, whether a surface format can be used as a texture and a render target, or whether a surface format can be used as a depth-stencil buffer. In addition, this method is used to verify depth buffer format support and depth-stencil buffer format support.

IDirect3D8::CheckDeviceType

Used to verify a device's ability to perform hardware acceleration, a device's ability to build a swap chain for presentation, or a device's ability to render to the current display format.

IDirect3D8::CheckDepthStencilMatch

Used to verify whether a depth-stencil buffer format is compatible with a render-target format. Note that before calling this method, the application should call **CheckDeviceFormat** on both the depth-stencil and render target formats.

[Visual Basic]

Direct3D8.CheckDeviceFormat

Used to verify whether a surface format can be used as a texture, whether a surface format can be used as a texture and a render target, or whether a surface format can be used as a depth-stencil buffer. In addition, this method is used to verify depth-buffer format support and depth-stencil buffer format support.

Direct3D8.CheckDeviceType

Used to verify a device's ability to perform hardware acceleration, a device's ability to build a swap chain for presentation, or a device's ability to render to the current display format.

Direct3D8.CheckDepthStencilMatch

Used to verify whether a depth-stencil buffer format is compatible with a render-target format. Note that before calling this method, the application should call **CheckDeviceFormat** on both the depth-stencil and render-target formats.

Selecting a Device

Applications can query hardware to detect the supported Microsoft® Direct3D® device types. This section contains information on the primary tasks involved in enumerating display adapters and selecting Direct3D devices.
An application must perform a series of tasks to select an appropriate Direct3D device. Note that the following steps are intended for a full-screen application. In most cases, a windowed application can skip most of these steps.

[C++]

1. Initially, the application must enumerate the display adapters on the system. An adapter is a physical piece of hardware. Note that the graphics card might contain more than a single adapter, as is the case with a dual-head display. Applications that are not concerned with multimonitor support can disregard this step, and pass D3DADAPTER_DEFAULT to the **IDirect3D8::EnumAdapterModes** method in step 2.
2. For each adapter, the application enumerates the supported display modes by calling **EnumAdapterModes**.
3. If required, the application checks for the presence of hardware acceleration in each enumerated display mode by calling **IDirect3D8::CheckDeviceType**, as shown in the following code example. Note that this is only one of the possible uses for **CheckDeviceType**; for details, see Determining Hardware Support.

```
D3DPRESENT_PARAMETERS Params;
// Initialize values for D3DPRESENT_PARAMETERS members.
```

```
Params.BackBufferFormat = D3DFMT_X1R5G5B5;

if(FAILED(m_pD3D->CheckDeviceType(Device.m_uAdapter,
                        Device.m_DevType,
                        Params.BackBufferFormat, Params.BackBufferFormat,
                        FALSE)))
    return E_FAIL;
```

4. The application checks for the desired level of functionality for the device on this adapter by calling the **IDirect3D8::GetDeviceCaps** method. This method filters out those devices that do not support the required functionality. The device capability returned by **GetDeviceCaps** is guaranteed to be constant for a device across all display modes verified by **CheckDeviceType**.
5. Devices can always render to surfaces of the format of an enumerated display mode that is supported by the device. If the application is required to render to a surface of a different format, it can call **IDirect3D8::CheckDeviceFormat**. If the device can render to the format, then it is guaranteed that all capabilities returned by **GetDeviceCaps** are applicable.
6. Lastly, the application can determine if multisampling techniques, such as full-scene antialiasing, are supported for a render format by using the **IDirect3D8::CheckDeviceMultiSampleType** method.

[Visual Basic]

1. Initially, the application must enumerate the display adapters on the system. An adapter is a physical piece of hardware. Note that the graphics card might contain more than a single adapter, as is the case with a dual-head display. Applications that are not concerned with multimonitor support can disregard this step, and pass D3DADAPTER_DEFAULT to the **Direct3D8.EnumAdapterModes** method in step 2.
2. For each adapter, the application enumerates the supported display modes by calling **EnumAdapterModes**.
3. If required, the application checks for the presence of hardware acceleration in each enumerated display mode by calling **Direct3D8.CheckDeviceType**. Note that this is only one of the possible uses for **CheckDeviceType**. For details see Determining Hardware Support.
4. The application checks for the desired level of functionality for the device on this adapter by calling the **Direct3D8.GetDeviceCaps** method. This method filters out those devices that do not support the required functionality. The device capability returned by **GetDeviceCaps** is guaranteed to be constant for a device across all display modes verified by **CheckDeviceType**.
5. Devices can always render to surfaces of the format of an enumerated display mode that is supported by the device. If the application is required to render to a surface of a different format, it can call **Direct3D8.CheckDeviceFormat**. If the

device can render to the format, then it is guaranteed that all capabilities returned by **GetDeviceCaps** are applicable.

6. Lastly, the application can determine if multisampling techniques, such as full-scene antialiasing, are supported for a render format by using the **Direct3D8.CheckDeviceMultiSampleType** method.

After completing the above steps, the application should have a list of display modes in which it can operate. The final step is to verify that enough device-accessible memory is available to accommodate the required number of buffers and antialiasing. This test is necessary because the memory consumption for the mode and multisample combination is impossible to predict without verifying it. Moreover, some display adapter architectures might not have a constant amount of device-accessible memory. This means that an application should be able to report out-of-video-memory failures when going into full-screen mode. Typically, an application should remove full-screen mode from the list of modes it offers to a user, or it should attempt to consume less memory by reducing the number of back buffers or using a less expensive multisampling technique.

[C++]
A windowed application performs a similar set of tasks.

1. It determines the desktop rectangle covered by the client area of the window.
2. It enumerates adapters, looking for the adapter whose monitor covers the client area. If the client area is owned by more than one adapter, then the application can choose to drive each adapter independently, or to drive a single adapter and have Direct3D transfer pixels from one device to another at presentation. The application can also disregard the above two steps and use the D3DADAPTER_DEFAULT adapter. Note that this might result in slower operation when the window is placed on a secondary monitor.
3. The application should call **CheckDeviceType** to determine if the device can support rendering to a back buffer of the specified format while in desktop mode. **IDirect3D8::GetAdapterDisplayMode** can be used to determine the desktop display format, as shown in the following code example.

```
D3DPRESENT_PARAMETERS Params;
// Initialize values for D3DPRESENT_PARAMETERS members.

// Use the current display mode.
D3DDISPLAYMODE mode;

if(FAILED(m_pD3D->GetAdapterDisplayMode(Device.m_uAdapter , &mode)))
    return E_FAIL;

Params.BackBufferFormat = mode.Format;

if(FAILED(m_pD3D->CheckDeviceType(Device.m_uAdapter, Device.m_DevType,
Params.BackBufferFormat, Params.BackBufferFormat, FALSE)))
```

```
return E_FAIL;
```

[Visual Basic]
A windowed application performs a similar set of tasks.

1. It determines the desktop rectangle covered by the client area of the window.
2. It enumerates adapters, looking for the adapter whose monitor covers the client area. If the client area is owned by more than one adapter, then the application can choose to drive each adapter independently, or to drive a single adapter and have Direct3D transfer pixels from one device to another at presentation. The application can also disregard steps one and two and use the D3DADAPTER_DEFAULT adapter. Note that this might result in slower operation when the window is placed on a secondary monitor.
3. The application should call **CheckDeviceType** to determine if the device can support rendering to a back buffer of the specified format while in desktop mode. **Direct3D8.GetAdapterDisplayMode** can be used to determine the desktop display format.

Creating a Device

[C++]

Note

All rendering devices created by a given Microsoft® Direct3D® object share the same physical resources. Although your application can create multiple rendering devices from a single Direct3D object, because they share the same hardware, extreme performance penalties will be incurred.

To create a Direct3D device in a C++ application, your application must first create a Direct3D object, as explained in Accessing Direct3D.
First, initialize values for the **D3DPRESENT_PARAMETERS** structure that is used to create the Direct3D device. The following code example specifies a windowed application where the back buffer is flipped to the front buffer on VSYNC only.

```
LPDIRECT3DDEVICE8 d3dDevice = NULL;

D3DPRESENT_PARAMETERS d3dpp;

ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed   = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_COPY_VSYNC;
```

Next, create the Direct3D device. The following **IDirect3D8::CreateDevice** call specifies the default adapter, a hardware abstraction layer (HAL) device, and software vertex processing.

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                    &d3dpp, &d3dDevice ) ) )
    return E_FAIL;
```

Note that a call to create, release, or reset the device should happen only on the same thread as the window procedure of the focus window.
After creating the device, set its state.

[Visual Basic]

Note

All rendering devices created by a given Microsoft® Direct3D® object share the same physical resources. Although your application can create multiple rendering devices from a single Direct3D object, because they share the same hardware, extreme performance penalties will be incurred.

To create a Direct3D device in a Microsoft Visual Basic® application, your application must first create a Direct3D object, as explained in Accessing Direct3D. First, initialize values for the **D3DPRESENT_PARAMETERS** type that is used to create the Direct3D device. The following code example specifies a windowed application where the back buffer is flipped to the front buffer on VSYNC only.

```
Dim d3dDevice As Direct3DDevice8

    Dim d3dpp As D3DPRESENT_PARAMETERS

    d3dpp.Windowed = 1
    d3dpp.SwapEffect = D3DSWAPEFFECT_COPY_VSYNC
```

Next, create the Direct3D device. The following **Direct3D8.CreateDevice** call specifies the default adapter, a hardware abstraction layer (HAL) device, and software vertex processing.

```
    Set d3dDevice = g_D3D.CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
hWnd, _
                                D3DCREATE_SOFTWARE_VERTEXPROCESSING, d3dpp)
    If d3dDevice Is Nothing Then Exit Function
```

After creating the device, set its state.

Setting Transformations

[C++]

In C++, transformations are applied by using the **Direct3DDevice8.SetTransform** method. For example, you can use code like the following to set the view transformation.

```
HRESULT    hr;
D3DMATRIX  view;

// Fill in the view matrix.

hr = pDev->SetTransform(D3DTS_VIEW, &view);

if(FAILED(hr))
{
   // Code to handle the error goes here.
}
```

There are several possible settings for the first parameter in a call to **SetTransform**. The most common are D3DTS_WORLD, D3DTS_VIEW, and D3DTS_PROJECTION. These transformation states are defined by the **D3DTRANSFORMSTATETYPE** enumerated type. This enumeration also contains the states D3DTS_TEXTURE1 through D3DTS_TEXTURE7, which you can use to apply transformations to texture coordinates.

[Visual Basic]
In Microsoft® Visual Basic®, transformations are applied by using the **Direct3DDevice8.SetTransform** method. For example, you could use code like the following to set the view transformation.

```
On Local Error Resume Next
Dim view As D3DMATRIX

' Fill in the view matrix.

Call d3dDevice.SetTransform(D3DTS_VIEW, view)

If Err.Number <> DD_OK Then
   ' Code to handle the error goes here.
End If
```

There are several possible settings for the first parameter in a call to **SetTransform**. The most common are D3DTS_WORLD, D3DTS_VIEW, and D3DTS_PROJECTION. These transformation states are defined by the **CONST_D3DTRANSFORMSTATETYPE** enumeration. This enumeration also contains the states D3DTS_TEXTURE1 through D3DTS_TEXTURE7, which you can use to apply transformations to texture coordinates.

Processing Vertex Data

[C++]

The **IDirect3DDevice8** interface supports vertex processing in both software and hardware. In general, the device capabilities for software and hardware vertex processing are not identical. Hardware capabilities are variable, depending on the display adapter and driver, while software capabilities are fixed.
The following flags control vertex processing behavior for the hardware abstraction layer (HAL) and reference devices.

- D3DCREATE_SOFTWARE_VERTEXPROCESSING
- D3DCREATE_HARDWARE_VERTEXPROCESSING
- D3DCREATE_MIXED_VERTEXPROCESSING

Specify one of the vertex processing behavior flags when calling **IDirect3D8::CreateDevice**. The mixed-mode flag enables the device to perform both software and hardware vertex processing. Only one vertex processing flag may be set for a device at any one time. Note that the D3DCREATE_HARDWARE_VERTEXPROCESSING flag is required to be set when creating a pure device (D3DCREATE_PUREDEVICE).
To avoid dual vertex processing capabilities on a single device, only the hardware vertex processing capabilities can be queried at run time. Software vertex processing capabilities are fixed and cannot be queried at run time.
You can consult the **VertexProcessingCaps** member of the **D3DCAPS8** structure to determine the hardware vertex processing capabilities of the device. For software vertex processing the following capabilities are supported.

- D3DVTXPCAPS_DIRECTIONALLIGHTS
- D3DVTXPCAPS_LOCALVIEWER
- D3DVTXPCAPS_MATERIALSOURCE7
- D3DVTXPCAPS_POSITIONALLIGHTS
- D3DVTXPCAPS_TEXGEN
- D3DVTXPCAPS_TWEENING

In addition, the following table lists the values that are set for members of the **D3DCAPS8** structure for a device in software vertex processing mode.

Member	**Software Vertex Processing Capabilities**
MaxActiveLights	Unlimited
MaxUserClipPlanes	6
MaxVertexBlendMatrices	4
MaxStreams	16
MaxVertexIndex	0xFFFFFFFF

[Visual Basic]
The **Direct3DDevice8** class supports vertex processing in both software and hardware. In general, the device capabilities for software and hardware vertex

processing are not identical. Hardware capabilities are variable, depending on the display adapter and driver, while software capabilities are fixed.
The following flags control vertex processing behavior for the hardware abstraction layer (HAL) and reference devices.

- D3DCREATE_SOFTWARE_VERTEXPROCESSING
- D3DCREATE_HARDWARE_VERTEXPROCESSING
- D3DCREATE_MIXED_MODE_VERTEXPROCESSING

Specify one of the vertex processing behavior flags when calling **Direct3D8.CreateDevice**. The mixed-mode flag enables the device to perform both software and hardware vertex processing. Only one vertex processing flag may be set for a device at any one time. Note that the D3DCREATE_HARDWARE_VERTEXPROCESSING flag is required to be set when creating a pure device (D3DCREATE_PUREDEVICE).
To avoid dual vertex processing capabilities on a single device, only the hardware vertex processing capabilities can be queried at run time. Software vertex processing capabilities are fixed and cannot be queried at run time.
You can consult the **VertexProcessingCaps** member of the **D3DCAPS8** structure to determine the hardware vertex processing capabilities of the device. For software vertex processing the following capabilities are supported.

- D3DVTXPCAPS_DIRECTIONALLIGHTS
- D3DVTXPCAPS_LOCALVIEWER
- D3DVTXPCAPS_MATERIALSOURCE7
- D3DVTXPCAPS_POSITIONALLIGHTS
- D3DVTXPCAPS_TEXGEN
- D3DVTXPCAPS_TWEENING

In the addition, the following table lists the values that are set for members of the **D3DCAPS8** structure for a device in software vertex processing mode.

Member	**Software Vertex Processing Capabilities**
MaxActiveLights	Unlimited
MaxUserClipPlanes	6
MaxVertexBlendMatrices	4
MaxStreams	16
MaxVertexIndex	0xFFFFFFFF

Software vertex processing provides a guaranteed set of vertex processing capabilities, including an unbounded number of lights and full support for programmable vertex shaders. You can toggle between software and hardware vertex processing at any time when using the HAL Device, the only device type that supports both hardware and software vertex processing. The only requirement is that

vertex buffers used for software vertex processing must be allocated in system memory.

Note

The performance of hardware vertex processing is comparable to that of software vertex processing. For this reason, it's a good idea to provide, within a single device type, both hardware- and software-emulation functionality for vertex processing. This is not the case for rasterization, for which host processors are much slower than specialized graphics hardware. Thus both hardware- and software-emulated rasterization is not provided within a single device type. Software vertex processing is the only instance of functionality duplicated between the run time and the hardware (driver) within a single device. Thus all other device capabilities represent potentially variable functionality provided by the driver.

Working with Mouse Cursors

The mouse cursor methods enable the application to specify a color cursor by providing a surface that contains an image. The system will ensure that this cursor will be updated at half the display rate or more if the application's frame rate is slow. However, the cursor will never be updated more frequently than the display refresh rate.

[C++]
The mouse cursor position is tied to the system cursor, appropriately scaled for the current display mode spatial resolution, but it can be moved explicitly by the application. This is analogous to the behavior of the Microsoft® Win32® API–supported system mouse cursor. For more information on how to use a mouse cursor in your Microsoft® Direct3D® application, see the following reference topics.

- **IDirect3DDevice8::ShowCursor**
- **IDirect3DDevice8::SetCursorPosition**
- **IDirect3DDevice8::SetCursorProperties**

Direct3D ensures that the mouse is supported either by hardware implementations or by the Direct3D run time that performs hardware-accelerated blitting operations when calling **IDirect3DDevice8::Present**.

[Visual Basic]
The mouse cursor position is tied to the system cursor (appropriately scaled for the current display mode spatial resolution) but it can be moved explicitly by the application. This is analogous to the behavior of the Microsoft® Win32® API supported system mouse cursor. For more information on how to use a mouse cursor in your Microsoft® Direct3D® application, see the following reference topics.

- **Direct3DDevice8.ShowCursor**
- **Direct3DDevice8.SetCursorPosition**
- **Direct3DDevice8.SetCursorProperties**

Direct3D ensures that the mouse will be supported either by hardware implementations or by the Direct3D run time performing hardware-accelerated blitting operations when calling **Direct3DDevice8.Present**.

Rendering

Applications use the DrawPrimitive family of methods to render a 3-D scene. The following topics discuss rendering with DrawPrimitive.

- Clearing Surfaces
- Beginning and Ending a Scene
- Presenting a Scene
- Rendering Primitives

Clearing Surfaces

Before rendering objects in a scene, clear the viewport on the render-target surface or a subset of the viewport. Clearing the viewport causes the system to set the desired portion of the render-target surface and any attached depth or stencil buffers to a desired state. This resets the areas of the surface that will be rendered again, and it resets the corresponding areas of the depth and stencil buffers, if any are in use. Clearing a render-target surface can set the desired region to a default color or texture. For depth and stencil buffers, this can set a depth or stencil value.

[C++]
Use the **IDirect3DDevice8::Clear** method to clear the viewport.
For more information about using this method, see Clearing a Viewport.

[Visual Basic]
Use the **Direct3DDevice8.Clear** method to clear the viewport.
For more information about using this method, see Clearing a Viewport.

Optimization Note

Applications that render scenes covering the entire area of the render-target surface can improve performance by clearing the attached depth and stencil buffer surfaces, if any, instead of the render target. In this case, clearing the depth buffer causes Microsoft® Direct3D® to rewrite the render target on the next rendered frame, making an explicit clear operation on the render target redundant. However, if your application renders only to a portion of the render-target surface, explicit clear operations are required.

Beginning and Ending a Scene

[C++]

Applications written in C++ notify Microsoft® Direct3D® that scene rendering is about to begin by calling the **IDirect3DDevice8::BeginScene** method. **BeginScene** causes the system to check its internal data structures and the availability and validity of rendering surfaces. It also sets an internal flag to signal that a scene is in progress. After you begin a scene, you can call the various rendering methods to render the primitives or individual vertices that make up the objects in the scene. Attempts to call rendering methods when a scene is not in progress fail. For more information, see Rendering Primitives.
After you complete the scene, call the **IDirect3DDevice8::EndScene** method. The **EndScene** method flushes cached data, verifies the integrity of rendering surfaces, and clears an internal flag that signals when a scene is in progress.
All rendering methods must be bracketed by calls to the **BeginScene** and **EndScene** methods. If surfaces are lost or internal errors occur, the scene methods return error values. If **BeginScene** fails, the scene does not begin, and subsequent calls to **EndScene** will fail.
To summarize these cases:

- If **BeginScene** returns any error, you must not call **EndScene** because the scene has not successfully begun.
- If **BeginScene** succeeds, but you get an error while rendering the scene, you must call **EndScene**.
- If **EndScene** fails, for any reason, you need not call it again.

For example, some simple scene code might look like the following example.

```
HRESULT hr;

if(SUCCEEDED(pDevice->BeginScene()))
{
    // Render primitives only if the scene
    // starts successfully.

    // Close the scene.
    hr = pDevice->EndScene();
    if(FAILED(hr))
        return hr;
}
```

[Visual Basic]
Microsoft® Visual Basic® applications notify Microsoft Direct3D® that scene rendering is about to begin by calling the **Direct3DDevice8.BeginScene** method. **BeginScene** causes the system to check its internal data structures and the availability and validity of rendering surfaces. It also sets an internal flag to signal that a scene is in progress. After you begin a scene, you can call the various rendering methods to render the primitives or individual vertices that make up the objects in the scene.

Attempts to call rendering methods when a scene is not in progress fail. For more information, see Rendering Primitives.
After you complete the scene, call the **Direct3DDevice8.EndScene** method. The **EndScene** method flushes cached data, verifies the integrity of rendering surfaces, and clears an internal flag that signals when a scene is in progress.
All rendering methods must be bracketed by calls to the **BeginScene** and **EndScene** methods. If surfaces are lost or internal errors occur, the scene methods fail and **Err.Number** is set to an error code. If **BeginScene** fails, the scene does not begin, and subsequent calls to **EndScene** will fail. When surfaces are lost during rendering, **EndScene** will fail.
To summarize these cases:

- If **BeginScene** returns any error, you must not call **EndScene** because the scene has not successfully begun.
- If **BeginScene** succeeds, but you get an error while rendering the scene, you must call **EndScene**.
- If **EndScene** fails, for any reason, you need not call it again.

For example, some simple scene code might look like the following example.

```
On Local Error Resume Next

Call d3dDevice.BeginScene
If Err.Number = DD_OK Then
   ' Render primitives only if the scene
   ' starts successfully.

   ' Close the scene.
   Call d3dDevice.EndScene
   If Err.Number <> DD_OK Then
      ' Code to handle error goes here.
   End If
End If
```

You cannot embed scenes; that is, you must complete rendering a scene before you can begin another one. Calling **EndScene** when **BeginScene** has not been called returns an error value. Likewise, calling **BeginScene** when a previous scene has not been completed with the **EndScene** method results in an error.
Do not attempt to call GDI functions on Direct3D surfaces, such as the render target or textures, while a scene is being rendered—is between **BeginScene** and **EndScene** calls. Attempts to do so can prevent the results of the GDI operations from being visible. If your application uses GDI functions, be sure that all GDI calls are made outside the scene functions.

Presenting a Scene

This section introduces the presentation application programming interfaces (APIs) and discusses the issues involved in presenting a scene to the display.
Information is divided into the following topics.

- Introduction to Presenting a Scene
- Multiple Views in Windowed Mode
- Multiple-Monitor Operations
- Manipulating the Depth Buffer
- Accessing the Color Front Buffer

Introduction to Presenting a Scene

[C++]
The presentation application programming interfaces (APIs) are a set of methods that control the state of the device that affects what the user sees on the monitor. These methods include setting display modes and once-per-frame methods that are used to present images to the user.

- **IDirect3DDevice8::Present**
- **IDirect3DDevice8::Reset**
- **IDirect3DDevice8::GetGammaRamp**
- **IDirect3DDevice8::SetGammaRamp**
- **IDirect3DDevice8::GetRasterStatus**

Familiarity with the following terms is necessary to understand the presentation APIs.

- *Front Buffer*. A rectangle of memory that is translated by the graphics adapter and displayed on the monitor or other output device.
- *Back Buffer*. A surface whose contents can be promoted to the front buffer.
- *Swap Chain*. A collection of back buffers that can be serially presented to the front buffer. Typically, a full-screen swap chain presents subsequent images with the flipping DDI, and a windowed swap chain presents images with the blitting DDI.

Because Microsoft® Direct3D® for Microsoft DirectX® 8.0 has one swap chain as a property of the device, there is always at least one swap chain per device. The **IDirect3DDevice8** interface has a set of methods that manipulate the implicit swap chain and are a copy of the swap chain's own interface. Applications can create additional swap chains; however, this is not necessary for the typical single window or full-screen application.
The front buffer is not directly exposed in the Direct3D API for DirectX 8.0. As a result, applications cannot lock or render to the front buffer. For details, see Accessing the Color Front Buffer.

Note

DirectX 7.*x* provided a number of presentation APIs that were called together. A good example of this is the **IDirectDraw7::SetCooperativeLevel**,

IDirectDraw7::SetDisplayMode, and **IDirectDraw7::CreateSurface** sequence. Additionally, the **IDirectDrawSurface7::Flip** and **IDirectDrawSurface7::Blt** methods signaled the transport of rendered frames to the monitor. DirectX 8.0 collapses these groups of APIs into two main methods, **Reset** and **Present**. **Reset** subsumes **SetCooperativeLevel**, **SetDisplayMode**, **CreateSurface**, and some of the parameters to **Flip**. **Present** subsumes **Flip** and the presentation uses of **Blt**.

A call to **IDirect3D8::CreateDevice** represents an implicit reset of the device. The DirectX 8.0 API has no notion of a primary surface; you cannot create an object that represents the primary surface. It is considered to be an internal property of the device.
Gamma ramps are associated with a swap chain and are manipulated with the **IDirect3DDevice8::GetGammaRamp** and **IDirect3DDevice8::SetGammaRamp** methods.

[Visual Basic]
The presentation application programming interfaces (APIs) are a set of methods that control the state of the device that affects what you see on the monitor. These methods include setting display modes and once-per-frame methods that are used to present images to the user.

- **Direct3DDevice8.Present**
- **Direct3DDevice8.Reset**
- **Direct3DDevice8.GetGammaRamp**
- **Direct3DDevice8.SetGammaRamp**
- **Direct3DDevice8.GetRasterStatus**

Familiarity with the following terms is necessary to understand the presentation APIs.

- *Front Buffer*. A rectangle of memory that is translated by the graphics adapter and displayed on the monitor or other output device.
- *Back Buffer*. A surface whose contents can be promoted to the front buffer.
- *Swap Chain*. A collection of back buffers that can be serially presented to the front buffer. Typically, a full-screen swap chain presents subsequent images with the flipping DDI, and a windowed swap chain presents images with the blitting DDI.

Because Microsoft® Direct3D® Microsoft for DirectX® 8.0 has one swap chain as a property of the device, there is always at least one swap chain per device. The **Direct3DDevice8** class has a set of methods that manipulate the implicit swap chain and are a copy of the swap chain's own interface. Applications can create additional swap chains; however, this is not necessary for the typical single window or full-screen application.
The front buffer is not directly exposed in the Direct3D API for DirectX 8.0. As a result, applications cannot lock or render to the front buffer. For details, see Accessing the Color Front Buffer.

Note

DirectX 7.*x* provided a number of presentation APIs that were called together. A good example of this is the **DirectDraw7.SetCooperativeLevel**, **DirectDraw7.SetDisplayMode**, and **DirectDraw7.CreateSurface** sequence. Additionally, the **DirectDrawSurface7.Flip** and **DirectDrawSurface7.Blt** methods signaled the transport of rendered frames to the monitor. DirectX 8.0 collapses these groups of APIs into two main methods, **Reset** and **Present**. **Reset** subsumes **SetCooperativeLevel**, **SetDisplayMode**, **CreateSurface**, and some of the parameters to **Flip**. **Present** subsumes **Flip** and the presentation uses of **Blt**.

A call to **Direct3D8.CreateDevice** represents an implicit reset of the device. The DirectX 8.0 API has no notion of a primary surface; you cannot to create an object that represents the primary surface. It is considered to be an internal property of the device.
Gamma ramps are associated with a swap chain and are manipulated with the **Direct3DDevice8.GetGammaRamp** and **Direct3DDevice8.SetGammaRamp** methods.

Multiple Views in Windowed Mode

[C++]
In addition to the swap chain that is owned and manipulated through the Direct3DDevice object, an application can use the **IDirect3DDevice8::CreateAdditionalSwapChain** method to create additional swap chains to present multiple views from the same device.
Typically, the application creates one swap chain per view, and it associates each swap chain with a particular view. The application renders images in the back buffers of each swap chain, and then uses the **IDirect3DDevice8::Present** method to present them individually. Note that only one swap chain at a time can be full-screen on each adapter.

[Visual Basic]
In addition to the swap chain that is owned and manipulated through the Direct3DDevice object, an application can use the **Direct3DDevice8.CreateAdditionalSwapChain** method to create additional swap chains to present multiple views from the same device.
Typically, the application creates one swap chain per view, and it associates each swap chain with a particular view. The application renders images in the back buffers of each swap chain, and then uses the **Direct3DDevice8.Present** method to present them individually. Note that only one swap chain at a time can be full-screen on each adapter.

Multiple-Monitor Operations

[C++]

When a device is successfully reset (**IDirect3DDevice8::Reset**) or created (**IDirect3D8::CreateDevice**) in full-screen operations, the Microsoft® Direct3D® object that created the device is marked as owning all adapters on that system. This state is known as exclusive mode, and the Direct3D object owns exclusive mode. Exclusive mode means that devices created by any other Direct3D object can neither assume full-screen operations nor allocate video memory. In addition, when a Direct3D object assumes exclusive mode, all devices other than the one that went full-screen are placed in lost state. For details, see Lost Devices.

[Visual Basic]
When a device is successfully reset (**Direct3DDevice8.Reset**) or created (**Direct3D8.CreateDevice**) in full-screen operations, the Microsoft® Direct3D® object that created the device is marked as owning all adapters on that system. This state is known as exclusive mode, and the Direct3D object owns exclusive mode. Exclusive mode means that devices created by any other Direct3D object can neither assume full-screen operations nor allocate video memory. In addition, when a Direct3D object assumes exclusive mode, all devices other than the one that went full-screen are placed in lost state. For details, see Lost Devices.

Along with exclusive mode, the Direct3D object is informed of the focus window that the device will use. Exclusive mode is released when the final full-screen device owned by that Direct3D object is either reset to windowed mode or destroyed. Devices can be divided into two categories when a Direct3D object owns exclusive mode. The first category of devices have the following characteristics.

- They are created by the same Direct3D object that created the device that is full-screen.
- They have the same focus window as the device that is full-screen.
- They represent a different adapter from any full-screen device.

Devices in this category have no restrictions concerning their ability to be reset or created, and they are not placed in lost state. Devices in this category can even be put into full-screen mode.
Devices that do not fall in the first category—devices created by another Direct3D object, created with a different focus window, and created for an adapter with a device that is already full-screen—cannot be reset and remain in lost state until the exclusive mode is lost. As a result, a multiple-monitor application can place several devices in full-screen mode, but only if all these devices are for different adapters, were created by the same Direct3D object, and share the same focus window.

Manipulating the Depth Buffer

[C++]
Depth buffers are associated with the device. Applications are required to move the depth buffers when they set render targets. The **IDirect3DDevice8::GetDepthStencilSurface** and **IDirect3DDevice8::SetRenderTarget** methods are used to manipulate depth buffers.

[Visual Basic]
Depth buffers are associated with the device. Applications are required to move the depth buffers when they set render targets. The **Direct3DDevice8.GetDepthStencilSurface** and **Direct3DDevice8.SetRenderTarget** methods are used to manipulate depth buffers.

Accessing the Color Front Buffer

[C++]
Accessing the front buffer is allowed through the **IDirect3DDevice8::GetFrontBuffer** method. Although this method is provided, it is intended explicitly for testing requirements. Thus, its functionality is not guaranteed.

[Visual Basic]
Accessing the front buffer is allowed through the **Direct3DDevice8.GetFrontBuffer** method. Although this method is provided, it is intended explicitly for testing requirements. Thus, its functionality is not guaranteed.

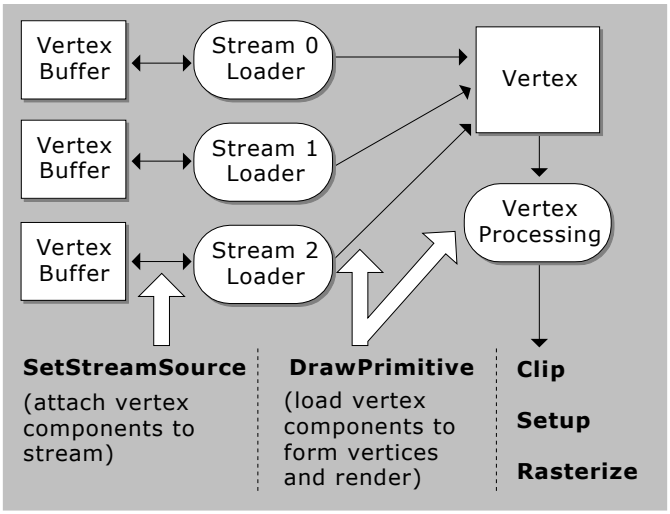
Rendering Primitives

The following topics introduce the **DrawPrimitive** rendering methods and provide information about using them in your application.

- Vertex Data Streams
- Setting the Stream Source
- Rendering from Vertex and Index Buffers
- Rendering from User Memory Pointers

Vertex Data Streams

The rendering interfaces for Microsoft® Direct3D® consist of methods that render primitives from vertex data stored in one or more data buffers. Vertex data consists of vertex elements combined to form vertex components. Vertex elements, the smallest unit of a vertex, represent entities such as position, normal, or color.
Vertex components are one or more vertex elements stored contiguously (interleaved per vertex) in a single memory buffer. A complete vertex consists of one or more components, where each component is in a separate memory buffer. To render a primitive, multiple vertex components are read and assembled so that complete vertices are available for vertex processing. The illustration below shows the process of rendering primitives using vertex components.



Rendering primitives consists of two steps. First, set up one or more vertex component streams; second, invoke a **DrawPrimitive** method to render from those streams. Identification of vertex elements within these component streams is specified by the vertex shader. For details, see Vertex Shaders.

The **DrawPrimitive** methods specify an offset in the vertex data streams so that an arbitrary contiguous subset of the primitives within one set of vertex data can be rendered with each draw invocation. This enables you to change the device rendering state between groups of primitives that are rendered from the same vertex buffers. Both indexed and nonindexed drawing methods are supported. For more information, see Rendering from Vertex and Index Buffers.

A secondary set of rendering interfaces supports passing vertex and index data directly from user memory pointers. For more information, see Rendering from User Memory Pointers.

Setting the Stream Source

[C++]

The **IDirect3DDevice8::SetStreamSource** method binds a vertex buffer to a device data stream, creating an association between the vertex data and one of several data stream ports that feed the primitive processing functions. The actual references to the stream data do not occur until a drawing method, such as **IDirect3DDevice8::DrawPrimitive**, is called.

A stream is defined as a uniform array of component data, where each component consists of one or more elements representing a single entity such as position, normal, color, and so on. The *Stride* parameter specifies the size of the component, in bytes.

[Visual Basic]

The **Direct3DDevice8.SetStreamSource** method binds a vertex buffer to a device data stream, creating an association between the vertex data and one of several data stream ports that feed the primitive processing functions. The actual references to the

stream data do not occur until a drawing method, such as **Direct3DDevice8.DrawPrimitive**, is called.
A stream is defined as a uniform array of component data, where each component consists of one or more elements representing a single entity such as position, normal, color, and so on. The *Stride* parameter specifies the size of the component, in bytes.

Rendering from Vertex and Index Buffers

[C++]
Both indexed and nonindexed drawing methods are supported by Microsoft® Direct3D®. The indexed methods use a single set of indices for all vertex components. Vertex data is presented to the Direct3D application programming interface (API) in vertex buffers, and index data for the indexed drawing methods is presented in index buffers. For more information, see the following reference topics.

- **IDirect3DDevice8::DrawPrimitive**
- **IDirect3DDevice8::DrawIndexedPrimitive**

These methods draw primitives from the current set of data input streams. For details on setting the current index array to an index buffer, see the **IDirect3DDevice8::SetIndices** method.

[Visual Basic]
Both indexed and nonindexed drawing methods are supported by Microsoft® Direct3D®. The indexed methods use a single set of indices for all vertex components. Vertex data is presented to the Direct3D application programming interface (API) in vertex buffers, and index data for the indexed drawing methods is presented in index buffers. For more information, see the following reference topics.

- **Direct3DDevice8.DrawPrimitive**
- **Direct3DDevice8.DrawIndexedPrimitive**

These methods draw primitives from the current set of data input streams. For details on setting the current index array to an index buffer, see the **Direct3DDevice8.SetIndices** method.

Rendering from User Memory Pointers

[C++]
A secondary set of rendering interfaces supports passing vertex and index data directly from user memory pointers. These interfaces support a single stream of vertex data only. For more information, see the following reference topics.

- **IDirect3DDevice8::DrawPrimitiveUP**
- **IDirect3DDevice8::DrawIndexedPrimitiveUP**

These methods render with data specified by user memory pointers, instead of vertex and index buffers.

[Visual Basic]
A secondary set of rendering interfaces supports passing vertex and index data directly from user memory pointers. These interfaces support a single stream of vertex data only. For more information, see the following reference topics.

- **Direct3DDevice8.DrawPrimitiveUP**
- **Direct3DDevice8.DrawIndexedPrimitiveUP**

These methods render with data specified by user memory pointers, instead of vertex and index buffers.

Device States

A Microsoft® Direct3D® device is a state computer; applications set up the state of the lighting, rendering, and transformation modules and then pass data through them during rendering.
This section describes render states, provides details about each render state used in Direct3D, and contains information about device state blocks, which applications can use to manipulate groups of devices states in a single call. The following topics are discussed.

- Render States
- Texture Stage States
- State Blocks

Render States

This section introduces the concept of render states and discusses the various render states in detail. Information in this section is organized into the following topics.

- About Render States
- Alpha Blending State
- Alpha Testing State
- Ambient Lighting State
- Antialiasing State
- Culling State
- Depth Buffering State
- Fog State
- Lighting State
- Outline and Fill State
- Per-Vertex Color State
- Primitive Clipping State

- Shading State
- Stencil Buffer State
- Texture Wrapping State

About Render States

Device render states control the behavior of the Microsoft® Direct3D® device's rasterization module. They do this by altering the attributes of the rendering state, what type of shading is used, fog attributes, and other rasterizer operations.

[C++]
Applications written in C++ control the characteristics of the rendering state by invoking the **IDirect3DDevice8::SetRenderState** method. The **D3DRENDERSTATETYPE** enumerated type specifies all possible rendering states. Your application passes a value from the **D3DRENDERSTATETYPE** enumeration as the first parameter to the **SetRenderState** method.
Fixed function vertex processing is controlled by the **SetRenderState** method and the following device render states. Most of these controls do not have any effect when using programmed vertex shaders.

- D3DRS_SPECULARENABLE
- D3DRS_FOGSTART
- D3DRS_FOGEND
- D3DRS_FOGDENSITY
- D3DRS_RANGEFOGENABLE
- D3DRS_LIGHTING
- D3DRS_AMBIENT
- D3DRS_FOGVERTEXMODE
- D3DRS_COLORVERTEX
- D3DRS_LOCALVIEWER
- D3DRS_NORMALIZENORMALS
- D3DRS_DIFFUSEMATERIALSOURCE
- D3DRS_SPECULARMATERIALSOURCE
- D3DRS_AMBIENTMATERIALSOURCE
- D3DRS_EMISSIVEMATERIALSOURCE
- D3DRS_VERTEXBLEND

In addition, the fixed-function vertex processing pipeline uses the following methods to set transforms, materials, and lights.

- **IDirect3DDevice8::SetTransform**
- **IDirect3DDevice8::SetMaterial**
- **IDirect3DDevice8::SetLight**
- **IDirect3DDevice8::LightEnable**

Note

D3DRS_SPECULARENABLE controls the addition of specular color in the pixel pipeline. D3DRS_FOGSTART, D3DRS_FOGEND, and D3DRS_FOGDENSITY control the start, end, and density of pixel fog density computation.

[Visual Basic]

Microsoft® Visual Basic® applications control the characteristics of the rendering state by invoking the **Direct3DDevice8.SetRenderState** method. The **CONST_D3DRENDERSTATETYPE** enumeration specifies all possible rendering states. Your application passes a value from the **CONST_D3DRENDERSTATETYPE** enumeration as the first parameter to the **SetRenderState** method.

Fixed function vertex processing is controlled by the **SetRenderState** method and the following device render states. Most of these controls do not have any effect when using programmed vertex shaders.

- D3DRS_SPECULARENABLE
- D3DRS_FOGSTART
- D3DRS_FOGEND
- D3DRS_FOGDENSITY
- D3DRS_RANGEFOGENABLE
- D3DRS_LIGHTING
- D3DRS_AMBIENT
- D3DRS_FOGVERTEXMODE
- D3DRS_COLORVERTEX
- D3DRS_LOCALVIEWER
- D3DRS_NORMALIZENORMALS
- D3DRS_DIFFUSEMATERIALSOURCE
- D3DRS_SPECULARMATERIALSOURCE
- D3DRS_AMBIENTMATERIALSOURCE
- D3DRS_EMISSIVEMATERIALSOURCE
- D3DRS_VERTEXBLEND

In addition, the fixed-function vertex processing pipeline uses the following methods to set transforms, materials, and lights.

- **Direct3DDevice8.SetTransform**
- **Direct3DDevice8.SetMaterial**
- **Direct3DDevice8.SetLight**
- **Direct3DDevice8.LightEnable**

Note

D3DRS_SPECULARENABLE controls the addition of specular color in the pixel pipeline. D3DRS_FOGSTART, D3DRS_FOGEND, and D3DRS_FOGDENSITY control the start, end, and density of pixel fog density computation.

Alpha Blending State

The alpha value of a color controls its transparency. Enabling alpha blending allows colors, materials, and textures on a surface to be blended with transparency onto another surface.
For more information, see Alpha Texture Blending and Multiple Texture Blending.

[C++]
Applications written in C++ use the D3DRS_ALPHABLENDENABLE render state to enable alpha transparency blending. The Microsoft® Direct3D® API allows many types of alpha blending. However, it is important to note the user's 3-D hardware might not support all the blending states allowed by Direct3D.
The type of alpha blending that is done depends on the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states. Source and destination blend states are used in pairs. The following code example demonstrates how the source blend state is set to D3DBLEND_SRCCOLOR and the destination blend state is set to D3DBLEND_INVSRCCOLOR.

```
// This code example assumes that d3dDevice is a
// valid pointer to an IDirect3DDevice8 interface.

// Set the source blend state.
d3dDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);

// Set the destination blend state.
d3dDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCCOLOR);
```

As a result of the calls in the preceding code example, Direct3D performs a linear blend between the source color—the color of the primitive that is rendered at the current location—and the destination color—the color at the current location in the frame buffer. This gives an appearance similar to tinted glass. Some of the color of the destination object seems to be transmitted through the source object. The rest of the color appears to be absorbed.
Altering the source and destination blend states can give the appearance of emissive objects in a foggy or dusty atmosphere. For instance, if your application models flames, force fields, plasma beams, or similarly radiant objects in a foggy environment, set the source and destination blend states to D3DBLEND_ONE.
Another application of alpha blending is to control the lighting in a 3-D scene, also called light mapping. Setting the source blend state to D3DBLEND_ZERO and the destination blend state to D3DBLEND_SRCALPHA darkens a scene according to the

source alpha information. The source primitive is used as a light map that scales the contents of the frame buffer to darken it when appropriate. This produces monochrome light mapping.
You can achieve color light mapping by setting the source alpha blending state to D3DBLEND_ZERO and the destination blend state to D3DBLEND_SRCCOLOR.

[Visual Basic]
Microsoft® Visual Basic® applications use the D3DRS_ALPHABLENDENABLE render state to enable alpha transparency blending. The Microsoft® Direct3D® API allows many types of alpha blending. However, it is important to note the user's 3-D hardware might not support all the blending states allowed by Direct3D.
The type of alpha blending that is done depends on the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states. Source and destination blend states are used in pairs. The following code example demonstrates how the source blend state is set to D3DBLEND_SRCCOLOR and the destination blend state is set to D3DBLEND_INVSRCCOLOR.

```
' This code example assumes that d3dDevice contains
' a reference to a Direct3DDevice8 object.

' Set the source blend state.
d3dDevice.SetRenderState D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR

' Set the destination blend state.
d3dDevice.SetRenderState D3DRS_DESTBLEND, D3DBLEND_INVSRCCOLOR
```

As a result of the calls in the preceding code example, Direct3D performs a linear blend between the source color—the color of the primitive that is rendered at the current location—and the destination color—the color at the current location in the frame buffer. This gives an appearance similar to tinted glass. Some of the color of the destination object seems to be transmitted through the source object. The rest of the color appears to be absorbed.
Altering the source and destination blend states can give the appearance of emissive objects in a foggy or dusty atmosphere. For instance, if your application models flames, force fields, plasma beams, or similarly radiant objects in a foggy environment, set the source and destination blend states to D3DBLEND_ONE.
Another application of alpha blending is to control the lighting in a 3-D scene, also called light mapping. Setting the source blend state to D3DBLEND_ZERO and the destination blend state to D3DBLEND_SRCALPHA darkens a scene according to the source alpha information. The source primitive is used as a light map that scales the contents of the frame buffer to darken it when appropriate. This produces monochrome light mapping.
You can achieve color light mapping by setting the source alpha blending state to D3DBLEND_ZERO and the destination blend state to D3DBLEND_SRCCOLOR.

Alpha Testing State

[C++]
C++ applications can use alpha testing to control when pixels are written to the render-target surface. By using the D3DRS_ALPHATESTENABLE render state, your application sets the current Microsoft® Direct3D® device so that it tests each pixel according to an alpha test function. If the test succeeds, the pixel is written to the surface. If it does not, Direct3D ignores the pixel. Select the alpha test function with the D3DRS_ALPHAFUNC render state. Your application can set a reference alpha value for all pixels to compare against by using the D3DRS_ALPHAREF render state.
The most common use for alpha testing is to improve performance when rasterizing objects that are nearly transparent. If the color data being rasterized is more opaque than the color at a given pixel (D3DPCMPCAPS_GREATEREQUAL), then the pixel is written. Otherwise, the rasterizer ignores the pixel altogether, saving the processing required to blend the two colors. The following code example checks if a given comparison function is supported and, if so, it sets the comparison function parameters required to improve performance during rendering.

```
// This code example assumes that pCaps is a
// D3DCAPS8 structure that was filled with a
// previous call to IDirect3D8::GetDeviceCaps.
if (pCaps.AlphaCmpCaps & D3DPCMPCAPS_GREATEREQUAL)
{
   dev->SetRenderState(D3DRS_ALPHAREF, (DWORD)0x00000001);
   dev->SetRenderState(D3DRS_ALPHATESTENABLE, TRUE);
   dev->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);
}

// If the comparison is not supported, render anyway.
// The only drawback is no performance gain.
```

Not all hardware supports all alpha-testing features. You can check the device capabilities by calling the **IDirect3D8::GetDeviceCaps** method. After retrieving the device capabilities, check the **AlphaCmpCaps** member of the associated **D3DCAPS8** structure for the desired comparison function. If the **AlphaCmpCaps** member contains only the D3DPCMPCAPS_ALWAYS capability or only the D3DPCMPCAPS_NEVER capability, the driver does not support alpha tests.

[Visual Basic]
Applications written in Microsoft® Visual Basic® use alpha testing to control when pixels are written to the render-target surface. By using the D3DRS_ALPHATESTENABLE render state, your application sets the current Microsoft® Direct3D® device so that it tests each pixel according to an alpha test function. If the test succeeds, the pixel is written to the surface. If it does not, Direct3D ignores the pixel. Select the alpha test function with the D3DRS_ALPHAFUNC render state. Your application can set a reference alpha value for all pixels to compare against by using the D3DRS_ALPHAREF render state.

The most common use for alpha testing is to improve performance when rasterizing objects that are nearly transparent. If the color data being rasterized is more opaque than the color at a given pixel (D3DPCMPCAPS_GREATEREQUAL), then the pixel is written. Otherwise, the rasterizer ignores the pixel altogether, saving the processing required to blend the two colors. The following code example checks if a given comparison function is supported and, if so, it sets the comparison function parameters required to improve performance during rendering.

```
' This example assumes that Caps is a
' D3DCAPS8 type that was filled with a
' previous call to Direct3D8.GetDeviceCaps.
If (Caps.AlphaCmpCaps And D3DPCMPCAPS_GREATEREQUAL) Then
    Call dev.SetRenderState(D3DRS_ALPHAREF, &H1)
    Call dev.SetRenderState(D3DRS_ALPHATESTENABLE, True)
    Call dev.SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL)
End If

' If the comparison is not supported, render anyway.
' The only drawback is no performance gain.
```

Not all hardware supports all alpha-testing features. You can check the device capabilities by calling the **Direct3D8.GetDeviceCaps** method. After retrieving the device capabilities, check the **AlphaCmpCaps** member of the associated **D3DCAPS8** type for the desired comparison function. If the **AlphaCmpCaps** member contains only the D3DPCMPCAPS_ALWAYS capability or only the D3DPCMPCAPS_NEVER capability, the driver does not support alpha tests.

Ambient Lighting State

Ambient light is surrounding light that radiates from all directions.
For specific information on how Microsoft® Direct3D® uses ambient light, see Direct Light vs. Ambient Light, and Mathematics of Direct3D Lighting.

[C++]

A C++ application sets the color of ambient lighting by invoking the **IDirect3DDevice8::SetRenderState** method and passing the enumerated value D3DRS_AMBIENT as the first parameter. The second parameter is a color value. The default value is zero.

```
// This code example assumes that d3dDevice is a
// valid pointer to an IDirect3DDevice8 interface.

// Set the ambient light.
d3dDevice->SetRenderState(D3DRS_AMBIENT, 0x00202020);
```

[Visual Basic]

A Microsoft Visual Basic® application sets the color of ambient lighting by invoking the **Direct3DDevice8.SetRenderState** method and passing the enumerated value D3DRS_AMBIENT as the first parameter. The second parameter is a color value. The default value is zero.

```
' This code example assumes that D3DDevice contains a valid
' reference to a Direct3DDevice8 object.

' Set the ambient light.
Call D3DDevice.SetRenderState(D3DRS_AMBIENT, &H202020)
```

Antialiasing State

Antialiasing is a method of making lines and edges appear smoother on the screen. Microsoft® Direct3D® supports two antialiasing methods: edge antialiasing and full-scene antialiasing.
For details about these techniques, see Antialiasing.

[C++]
By default, Direct3D does not perform antialiasing. To enable edge-antialiasing, which requires a second rendering pass, set the D3DRS_EDGEANTIALIAS render state to TRUE. To disable it, set D3DRS_EDGEANTIALIAS to FALSE.
To enable full-scene antialiasing, set the D3DRS_MULTISAMPLEANTIALIAS render state to TRUE. To disable it, set D3DRS_MULTISAMPLEANTIALIAS to FALSE.

[Visual Basic]
By default, Direct3D does not perform antialiasing. To enable edge-antialiasing, which requires a second rendering pass, set the D3DRS_EDGEANTIALIAS render state to True. To disable it, set D3DRS_EDGEANTIALIAS to False.
To enable full-scene antialiasing, set the D3DRS_MULTISAMPLEANTIALIAS to True. To disable it, set D3DRS_MULTISAMPLEANTIALIAS to False.

Culling State

As Microsoft® Direct3D® renders primitives, it culls primitives that are facing away from the user.

[C++]
C++ applications set the culling mode by using the D3DRS_CULLMODE render state, which can be set to a member of the **D3DCULL** enumerated type. By default, Direct3D culls back faces with counterclockwise vertices.
The following code sample illustrates the process of setting the culling mode to cull back faces with clockwise vertices.

```
// This code example assumes that d3dDevice is a valid
// pointer to an IDirect3DDevice8 interface.
```

```
// Set the culling state.
d3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
```

[Visual Basic]
Microsoft Visual Basic® applications set the culling mode by using the D3DRS_CULLMODE render state, which can be set to a member of the **CONST_D3DCULL** enumeration. By default, Direct3D culls back faces with counterclockwise vertices.
The following code sample illustrates the process of setting the culling mode to cull back faces with clockwise vertices.

```
' This code example assumes that d3dDevice contains a valid
' reference to a Direct3DDevice8 object.

' Set the culling state.
Call d3dDevice.SetRenderState(D3DRENDERSTATE_CULLMODE, D3DCULL_CW)
```

Depth Buffering State

Depth buffering is a method of removing hidden lines and surfaces. By default, Microsoft® Direct3D® does not use depth buffering.
For a conceptual overview of depth buffers, see What Are Depth Buffers?.

[C++]
C++ applications update the depth-buffering state with the D3DRS_ZENABLE render state, using a member of the **D3DZBUFFERTYPE** enumeration to specify the new state value.
If your application needs to prevent Direct3D from writing to the depth buffer, it can use the D3DRS_ZWRITEENABLE enumerated value, specifying D3DZB_FALSE as the second parameter for the call to **IDirect3DDevice8::SetRenderState**.
The following code example shows how the depth-buffer state is set to enable z-buffering.

```
// This code example assumes that d3dDevice is a
// valid pointer to an IDirect3DDevice8 interface.

// Enable z-buffering.
d3dDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
```

Your application can also use the D3DRS_ZFUNC render state to control the comparison function that Direct3D uses when performing depth buffering.
Z-biasing is a method of displaying one surface in front of another, even if their depth values are the same. You can use this technique for a variety of effects. A common example is rendering shadows on walls. Both the shadow and the wall have the same depth value. However, you want your application to show the shadow on the wall. Giving a z-bias to the shadow makes Direct3D display them properly (see D3DRS_ZBIAS).

[Visual Basic]
Microsoft Visual Basic® applications update the depth-buffering state with the D3DRS_ZENABLE render state, using a member of the **CONST_D3DZBUFFERTYPE** enumeration to specify the new state value.
If your application needs to prevent Direct3D from writing to the depth buffer, it can use the D3DRS_ZWRITEENABLE render state, specifying D3DZB_FALSE as the second parameter for the call to **Direct3DDevice8.SetRenderState**.
The following Visual Basic code example shows how the depth-buffer state is set to enable z-buffering.

```
' This code example assumes that d3dDevice contains a valid
' reference to a Direct3DDevice8 object.

' Enable z-buffering.
Call d3dDevice.SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE)
```

Your application can also use the D3DRS_ZFUNC render state to control the comparison function that Direct3D uses when performing depth buffering.
Z-biasing is a method of displaying one surface in front of another, even if their depth values are the same. You can use this technique for a variety of effects. A common example is rendering shadows on walls. Both the shadow and the wall have the same depth value. However, you want your application to show the shadow on the wall. Giving a z-bias to the shadow makes Direct3D display them properly (see D3DRS_ZBIAS).

Fog State

Fog effects can give a 3-D scene greater realism. You can use fog effects for more than simulating fog. They can also decrease the clarity of a scene with distance. This mirrors what happens in the real world; as objects become more distant from the user, their detail is less distinct.
For more information about using fog in your application, see Fog.

[C++]
A C++ application controls fog through device rendering states. The **D3DRENDERSTATETYPE** enumerated type includes states to control whether pixel (table) or vertex fog is used, what color it is, the fog formula the system applies, and the parameters of the formula.
You enable fog by setting the D3DRS_FOGENABLE render state to TRUE. The fog color can be set to any color value by using the D3DRS_FOGCOLOR render state; the alpha component of the fog color is ignored, .
The D3DRS_FOGTABLEMODE and D3DRS_FOGVERTEXMODE render states control the fog formula applied for fog calculations, and they indirectly control which type of fog is applied. Both render states can be set to a member of the **D3DFOGMODE** enumerated type. Setting either render state to D3DFOG_NONE

disables pixel or vertex fog, respectively. If both render states are set to valid modes, the system applies only pixel fog effects.
The D3DRS_FOGSTART and D3DRS_FOGEND render states control fog formula parameters for the D3DFOG_LINEAR mode. The D3DRS_FOGDENSITY render state controls fog density in the exponential fog modes.

[Visual Basic]
A Microsoft® Visual Basic® application controls fog through device rendering states. The **CONST_D3DRENDERSTATETYPE** enumeration includes states to control whether pixel (table) or vertex fog is used, what color it is, the fog formula the system applies, and the parameters of the formula.
You enable fog by setting the D3DRS_FOGENABLE render state to True. The fog color can be set to any color value by using the D3DRS_FOGCOLOR render state; the alpha component of the fog color is ignored.
The D3DRS_FOGTABLEMODE and D3DRS_FOGVERTEXMODE render states control the fog formula applied for fog calculations, and they indirectly control which type of fog is applied. Both render states can be set to a member of the **CONST_D3DFOGMODE** enumeration. Setting either render state to D3DFOG_NONE disables pixel or vertex fog, respectively. If both render states are set to valid modes, the system applies only pixel fog effects.
The D3DRS_FOGSTART and D3DRS_FOGEND render states control fog formula parameters for the D3DFOG_LINEAR mode. The D3DRS_FOGDENSITY render state controls fog density in the exponential fog modes.

For more information, see Fog Parameters.

Lighting State

Applications that use the Microsoft® Direct3D® geometry pipeline can enable or disable lighting calculations. Only vertices that contain a vertex normal are properly lit; vertices with no normal will use a dot product of zero in all lighting computations. Therefore, a vertex that does not use a normal receives no light.
For more information, see Mathematics of Direct3D Lighting.

[C++]
Applications enable Direct3D lighting by setting the D3DRS_LIGHTING render state to TRUE, which is the default setting, and they disable Direct3D lighting by setting the render state to FALSE.
The lighting render state is entirely independent of lighting computations that can be performed on vertices within a vertex buffer. The **IDirect3DDevice8::ProcessVertices** method accepts its own flags to control lighting calculations during vertex processing.

[Visual Basic]
Applications enable Direct3D lighting by setting the D3DRS_LIGHTING render state to True, which is the default setting, and they disable Direct3D lighting by setting the render state to False.

The lighting render state is entirely independent of lighting computations that can be performed on vertices within a vertex buffer. The **Direct3DDevice8.ProcessVertices** method accepts its own flags to control lighting calculations during vertex processing.

Outline and Fill State

[C++]
Primitives that have no textures are rendered with the color specified by their material, or with the colors specified for the vertices, if any. You can select the method to fill them by specifying a value defined by the **D3DFILLMODE** enumerated type for the D3DRS_FILLMODE render state.
To enable dithering, your application must pass the D3DRS_DITHERENABLE enumerated value as the first parameter to **IDirect3DDevice8::SetRenderState**. It must set the second parameter to TRUE to enable dithering, and FALSE to disable it.
At times, drawing the last pixel in a line can cause unsightly overlap with surrounding primitives. You can control this using the D3DRS_LASTPIXEL enumerated value. However, do not alter this setting without some forethought. Under some conditions, suppressing the rendering of the last pixel can cause unsightly gaps between primitives.
By default, Microsoft® Direct3D® devices use a solid outline for primitives. The outline pattern can be changed using the **D3DLINEPATTERN** structure. For details, see the D3DRS_LINEPATTERN render state.

[Visual Basic]
Primitives that have no textures are rendered with the color specified by their material, or with the colors specified for the vertices, if any. You can select the method to fill them by specifying a value defined by the **CONST_D3DFILLMODE** enumeration for the D3DRS_FILLMODE render state.
To enable dithering, your application must pass the D3DRS_DITHERENABLE enumerated value as the first parameter to **Direct3DDevice8.SetRenderState**. It must set the second parameter to True to enable dithering, and False to disable it.
At times, drawing the last pixel in a line can cause unsightly overlap with surrounding primitives. You can control this using the D3DRS_LASTPIXEL enumerated value. However, do not alter this setting without some forethought. Under some conditions, suppressing the rendering of the last pixel can cause unsightly gaps between primitives.
By default, Microsoft® Direct3D® devices use a solid outline for primitives. The outline pattern can be changed using the **D3DLINEPATTERN** type. For details, see the D3DRS_LINEPATTERN render state.

Per-Vertex Color State

[C++]

When using flexible vertex format (FVF) codes, vertices can contain both vertex color and vertex normal information. By default, Microsoft® Direct3D® uses this information when it calculates lighting. To disable use of vertex color lighting information, invoke the **IDirect3DDevice8::SetRenderState** method and pass D3DRS_COLORVERTEX as the first parameter. Set the second parameter to FALSE to disable vertex color lighting, or TRUE to enable it.
If per-vertex color is enabled, applications can configure the source from which the system retrieves color information for a vertex. The D3DRS_AMBIENTMATERIALSOURCE, D3DRS_DIFFUSEMATERIALSOURCE, D3DRS_EMISSIVEMATERIALSOURCE, and D3DRS_SPECULARMATERIALSOURCE render states control the ambient, diffuse, emissive, and specular color component sources, respectively. Each state can be set to members of the **D3DMATERIALCOLORSOURCE** enumerated type, which defines constants that instruct the system to use the current material, diffuse color, or specular color as the source for the specified color component.

[Visual Basic]
The following vertex types cannot contain both color and normal information.

- **D3DLVERTEX**
- **D3DLVERTEX2**
- **D3DTLVERTEX**
- **D3DTLVERTEX2**
- **D3DVERTEX**
- **D3DVERTEX2**

Because the default shading mode is Gouraud shading, which depends on both vertex color and normal information, Microsoft® Direct3D® does not use vertex color in lighting calculations involving the predefined vertex types.
However, when you use flexible vertex format (FVF) codes, vertices can contain both vertex color and vertex normal information. By default, Direct3D uses this information when it calculates lighting. To disable use of vertex color lighting information, invoke the **Direct3DDevice8.SetRenderState** method and pass D3DRS_COLORVERTEX as the first parameter. Set the second parameter to False to disable vertex color lighting, or True to enable it.
If per-vertex color is enabled, applications can configure the source from which the system retrieves color information for a vertex. The D3DRS_AMBIENTMATERIALSOURCE, D3DRS_DIFFUSEMATERIALSOURCE, D3DRS_EMISSIVEMATERIALSOURCE, and D3DRS_SPECULARMATERIALSOURCE render states control the ambient, diffuse, emissive, and specular color component sources, respectively. Each state can be set to members of the **CONST_D3DMATERIALCOLORSOURCE** enumeration, which defines constants that instruct the system to use the current

material, diffuse color, or specular color as the source for the specified color component.

Primitive Clipping State

[C++]
Microsoft® Direct3D® can clip primitives that render partially outside the viewport. When using C++, Direct3D clipping is controlled by the D3DRS_CLIPPING render state. Set this render state to TRUE (the default value) to enable primitive clipping. Set it to FALSE to disable Direct3D clipping services.
The primitive clipping render state is entirely independent of clipping operations that can be performed on vertices within a vertex buffer. The **IDirect3DDevice8::ProcessVertices** method accepts its own flags to control primitive clipping during vertex processing.

[Visual Basic]
Microsoft® Direct3D® can clip primitives that render partially outside the viewport. When using Microsoft Visual Basic®, Direct3D clipping is controlled by the D3DRS_CLIPPING render state. Set this render state to True (the default value) to enable primitive clipping. Set it to False to disable Direct3D clipping services.
The primitive clipping render state is entirely independent of clipping operations that can be performed on vertices within a vertex buffer. The **Direct3DDevice8.ProcessVertices** method accepts its own flags to control primitive clipping during vertex processing.

Shading State

[C++]
Microsoft® Direct3D® supports both flat and Gouraud shading. The default is Gouraud shading. To control the current shading mode, your C++ application specifies a member of the **D3DSHADEMODE** enumerated type for the D3DRS_SHADEMODE render state.
The following C++ code example demonstrates the process of setting the shading state to flat shading mode.

```
// This code example assumes that d3dDevice is a
// valid pointer to a IDirect3DDevice8 interface.

// Set the shading state.
d3dDevice->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
```

[Visual Basic]
Microsoft® Direct3D® supports both flat and Gouraud shading. The default is Gouraud shading. To control the current shading mode, your Microsoft Visual Basic®

application specifies a member of the **CONST_D3DSHADEMODE** enumeration for the D3DRS_SHADEMODE render state.
The following Visual Basic code example demonstrates the process of setting the shading state to flat shading mode.

```
' This example assumes that d3dDevice is a valid
' reference to a Direct3DDevice8 object.

' Set the shading state.
Call d3dDevice.SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT)
```

Stencil Buffer State

Applications use the stencil buffer to determine whether a pixel is written to the rendering target surface.
For details, see Stencil Buffers.

[C++]
Applications written in C++ enable or disable stenciling by calling the **IDirect3DDevice8::SetRenderState** method. Pass D3DRS_STENCILENABLE as the value of the first parameter. Set the value of the second parameter to TRUE or FALSE to enable or disable stenciling, respectively.
Set the comparison function that Microsoft® Direct3D® uses to perform the stencil test by calling **SetRenderState**. Set the value of the first parameter to D3DRS_STENCILFUNC. Pass a member of the **D3DCMPFUNC** enumerated type as the value of the second parameter.
The stencil reference value is the value in the stencil buffer that the stencil function uses for its test. By default, the stencil reference value is zero. Your application can set the value by calling **SetRenderState**. Pass D3DRS_STENCILREF as the value of the first parameter. Set the value of the second parameter to the new reference value.
Before the Direct3D module performs the stencil test for any pixel, it performs a bitwise **AND** operation of the stencil reference value and a stencil mask value. The result is compared against the contents of the stencil buffer using the stencil comparison function. Your application can set the stencil mask by calling **SetRenderState**. Pass D3DRS_STENCILMASK as the value of the first parameter. Set the value of the second parameter to the new stencil mask.
To set the action that Direct3D takes when the stencil test fails, call **SetRenderState** and pass D3DRS_STENCILFAIL as the first parameter. The second parameter must be a member of the **D3DSTENCILOP** enumerated type.
Your application can also control how Direct3D responds when the stencil test passes but the z-buffer test fails. Call **SetRenderState** and pass D3DRS_STENCILZFAIL as the first parameter and use a member of the **D3DSTENCILOP** enumerated type for the second parameter.
In addition, your application can control what Direct3D does when both the stencil test and the z-buffer test pass. Call **SetRenderState** and pass D3DRS_STENCILPASS as the first parameter. Again, the second parameter must be a member of the **D3DSTENCILOP** enumerated type.

[Visual Basic]
Applications written in Microsoft® Visual Basic® enable or disable stenciling by calling the **Direct3DDevice8.SetRenderState** method. Pass D3DRS_STENCILENABLE as the value of the first parameter. Set the value of the second parameter to True or False to enable or disable stenciling, respectively.
Set the comparison function that Microsoft® Direct3D® uses to perform the stencil test by calling **SetRenderState**. Set the value of the first parameter to D3DRS_STENCILFUNC. Pass a member of the **CONST_D3DCMPFUNC** enumeration as the value of the second parameter.
The stencil reference value is the value in the stencil buffer that the stencil function uses for its test. By default, the stencil reference value is zero. Your application can set the value by calling **SetRenderState**. Pass D3DRS_STENCILREF as the value of the first parameter. Set the value of the second parameter to the new reference value.
Before Direct3D module performs the stencil test for a pixel, it performs a bitwise **And** operation of the stencil reference value and a stencil mask value. The result is compared against the contents of the stencil buffer using the stencil comparison function. Your application can set the stencil mask by calling **SetRenderState**. Pass D3DRS_STENCILMASK as the value of the first parameter. Set the value of the second parameter to the new stencil mask.
To set the action that Direct3D takes when the stencil test fails, call **SetRenderState** and pass D3DRS_STENCILFAIL as the first parameter. The second parameter must be a member of the **CONST_D3DSTENCILOP** enumeration.
Your application can also control how Direct3D responds when the stencil test passes but the z-buffer test fails. Call **SetRenderState** and pass D3DRS_STENCILZFAIL as the first parameter and use a member of the **CONST_D3DSTENCILOP** enumeration for the second parameter.
In addition, your application can control what Direct3D does when both the stencil test and the z-buffer test pass. Call **SetRenderState** and pass D3DRS_STENCILPASS as the first parameter. Again, the second parameter must be a member of the **CONST_D3DSTENCILOP** enumeration.

Texture Wrapping State

[C++]
The D3DRS_WRAP0 through D3DRS_WRAP7 render states enable and disable u- and v-wrapping for various textures in the device's multitexture cascade. You can set these render states to a combination of the D3DWRAPCOORD_0, D3DWRAPCOORD_1, D3DWRAPCOORD_2, and D3DWRAPCOORD_3 flags to enable wrapping in first, second, third, and fourth directions of the texture. Use a value of zero to disable wrapping altogether. Texture wrapping is disabled in all directions for all texture stages by default.

[Visual Basic]
The D3DRS_WRAP0 through D3DRS_WRAP7 render states control texture wrapping for various textures in the device's multitexture cascade. You can set these

render states to a combination of the D3DWRAPCOORD_0, D3DWRAPCOORD_1, D3DWRAPCOORD_2, and D3DWRAPCOORD_3 values from the **CONST_D3DWRAPFLAGS** enumeration to enable wrapping in first, second, third, and fourth directions of the texture. Use a value of zero to disable wrapping altogether. Texture wrapping is disabled in all directions for all texture stages by default.

For a conceptual overview, see Texture Wrapping.

Texture Stage States

This section introduces the concept of texture stage states and discusses the various texture stage states in detail. Information in this section is organized into the following topics.

- About Texture Stage States
- Texture Addressing State
- Texture Border State
- Texture Filtering State
- Texture Blending State

About Texture Stage States

Texture stage states control the style of texturing and how texture filtering is done.

C++]
Applications written in C++ control the characteristics of the texture-related render states by invoking the **IDirect3DDevice8::SetTextureStageState** method. The **D3DTEXTURESTAGESTATETYPE** enumerated type specifies all the possible texture-related rendering states. Your application passes a value from the **D3DTEXTURESTAGESTATETYPE** enumeration as the first parameter to the **SetTextureStageState** method.
Applications set the texture for a stage by calling the **IDirect3DDevice8::SetTexture** method.

[Visual Basic]
Microsoft® Visual Basic® applications control the characteristics of the texture-related rendering states by invoking the **Direct3DDevice8.SetTextureStageState** method. The **CONST_D3DTEXTURESTAGESTATETYPE** enumeration specifies all possible texture-related rendering states. Your application passes a value from the **CONST_D3DRENDERSTATETYPE** enumeration as the first parameter to the **SetTextureStageState** method.
Applications set the texture for a stage by calling the **Direct3DDevice8.SetTexture** method.

Texture Addressing State

[C++]
C++ applications use the D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, and D3DTSS_ADDRESSW texture stage states to set the texture addressing.

[Visual Basic]
Microsoft® Visual Basic® applications use the D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, and D3DTSS_ADDRESSW texture stage states to set texture addressing.

Texture Border State

[C++]
C++ applications use the D3DTSS_BORDERCOLOR texture stage state to set border color texture addressing.

[Visual Basic]
Microsoft® Visual Basic® applications use the D3DTSS_BORDERCOLOR texture stage state to set border color texture addressing.

For more information, see Border Color Texture Address Mode.

Texture Filtering State

[C++]
C++ applications use the D3DTSS_MAGFILTER, D3DTSS_MINFILTER, and D3DTSS_MIPFILTER, and texture-stage states to control texture filtering.

[Visual Basic]
Microsoft® Visual Basic® applications use the D3DTSS_MAGFILTER, D3DTSS_MINFILTER, and D3DTSS_MIPFILTER, and texture-stage states to control texture filtering.

Texture Blending State

[C++]
C++ applications use the texture blending states defined by the D3DTSS_COLOROP and D3DTSS_ALPHAOP texture stage states to control texture blending.

[Visual Basic]

Microsoft® Visual Basic® applications use the texture blending states defined by the D3DTSS_COLOROP and D3DTSS_ALPHAOP texture stage states to control texture blending.

For more information, see Multiple Texture Blending.

State Blocks

The following topics in this section describe the key concepts of device state blocks and provide information about how applications use state blocks.

- About State Blocks
- Recording State Blocks
- Capturing State Blocks
- Applying State Blocks
- Creating Predefined State Blocks
- Deleting State Blocks

About State Blocks

A state block in Microsoft® Direct3D® is a group of device states—render states, lighting and material parameters, transformation states, texture stage states, and current texture information. The state block is a snapshot of the device's current state, or it is explicitly recorded. The snapshot can be applied to a device in a single call. Device-state blocks can be optimized by the rendering device to accelerate the common sequences of state changes that your application requires, or they can simply make applying device states easier.

[C++]
In C++, you receive a state-block handle when you finish recording a state block by calling the **IDirect3DDevice8::EndStateBlock** method, and when you capture a predefined set of device state data by calling the **IDirect3DDevice8::CreateStateBlock** method.

[Visual Basic]
In Microsoft Visual Basic®, you receive a state-block handle when you finish recording a state block by calling the **Direct3DDevice8.EndStateBlock** method, and when you capture a predefined set of device state data by calling the **Direct3DDevice8.CreateStateBlock** method.

Recording State Blocks

[C++]
The **IDirect3DDevice8** interface provides the **IDirect3DDevice8::BeginStateBlock** method to record device states in a state block as your application calls for them. The

BeginStateBlock method causes the system to start recording device state changes in a state block, rather than applying them to the device. After you call **BeginStateBlock**, calls to any of the following methods are recorded in a device state block.

- **IDirect3DDevice8::LightEnable**
- **IDirect3DDevice8::SetClipPlane**
- **IDirect3DDevice8::SetLight**
- **IDirect3DDevice8::SetMaterial**
- **IDirect3DDevice8::SetRenderState**
- **IDirect3DDevice8::SetTexture**
- **IDirect3DDevice8::SetTextureStageState**
- **IDirect3DDevice8::SetTransform**
- **IDirect3DDevice8::SetViewport**

When you're done recording the state block, notify the system to stop recording by calling the **IDirect3DDevice8::EndStateBlock** method. The **EndStateBlock** method places the handle of the state block in the variable whose address you pass in the *pToken* parameter. Your application uses this handle to apply the state block to the device as needed, to capture new state data into the block, and to delete the state block when it is no longer required.

[Visual Basic]
The **Direct3DDevice8** object offers the **Direct3DDevice8.BeginStateBlock** method to record device states in a state block as your application calls for them. The **BeginStateBlock** method causes the system to start recording device state changes in a state block, rather than applying them to the device. After you call **BeginStateBlock**, calls to any of the following methods are recorded in a device state block.

- **Direct3DDevice8.LightEnable**
- **Direct3DDevice8.SetClipPlane**
- **Direct3DDevice8.SetLight**
- **Direct3DDevice8.SetMaterial**
- **Direct3DDevice8.SetRenderState**
- **Direct3DDevice8.SetTexture**
- **Direct3DDevice8.SetTextureStageState**
- **Direct3DDevice8.SetTransform**
- **Direct3DDevice8.SetViewport**

When you are done recording the state block, notify the system to stop recording by calling the **Direct3DDevice8.EndStateBlock** method. The **EndStateBlock** method places the handle of the state block in the method's return value. Your application

uses this handle to apply the state block to the device as needed, to capture new state data into the block, and to delete the state block when it is no longer required.

Performance Note

For best results, group all state changes tightly between a **BeginStateBlock** and **EndStateBlock** pair.

It is important to check the error code from the **EndStateBlock** method. If the method fails, it is likely because the display mode has changed. Design your application to recover from this type of failure by recreating its surfaces and recording the state block again.

Capturing State Blocks

[C++]
The **IDirect3DDevice8::CaptureStateBlock** method updates the values within an existing state block to reflect the current state of the device. The method accepts a single parameter, *Token*, that identifies the state block that will receive the current state of the device if the call succeeds.
The **CaptureStateBlock** method is especially useful to update a state block that your application has already recorded to include slightly different state settings. To do so, apply the existing state block, change the necessary settings, then capture the state of the system back to the state block, as shown in the following code example.

```
// The token variable contains a handle to a device-state block for
// a previously recorded set of device states.

// Set the current state block.
d3dDevice->ApplyStateBlock(token);

// Change device states as needed.
.
.
.

// Capture the modified device state data back to the existing block.
d3dDevice->CaptureStateBlock(token);
```

The **CaptureStateBlock** method doesn't capture the entire state of the device; it only updates the values for the states already in the state block. For example, imagine a state block that contains the two operations shown in the following code example.

```
d3dDevice->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD);
d3dDevice->SetTexture(0, pTexture);
```

If your application changed either of these settings since the state block was originally recorded, the **CaptureStateBlock** method will update the states within the block—and only those states—to their new values.

[Visual Basic]
The **Direct3DDevice8.CaptureStateBlock** method updates the values in an existing state block to reflect the current state of the device. The method accepts a single parameter, *Token*, that identifies the state block that will receive the current state of the device if the call succeeds.
The **CaptureStateBlock** method is especially useful to update a state block that your application has already recorded to include slightly different state settings. To do so, apply the existing state block, change the necessary settings, then capture the state of the system back to the state block, as shown in the following code example.

```
' The Token variable contains a handle to a device state block for
' a previously recorded set of device states.

' Set the current state block.
Call d3dDevice.ApplyStateBlock(Token)

' Change device states as needed.
.
.
.

' Capture the modified device state data back to the existing block.
Call d3dDevice.CaptureStateBlock(Token)
```

The **CaptureStateBlock** method doesn't capture the entire state of the device; it only updates the values for the states already in the state block. For example, imagine a state block that contains the two operations shown in the following code example.

```
Call d3dDevice.SetRenderState(D3DRS_SHADEMODE, D3DSHADE_GOURAUD)
Call d3dDevice.SetTexture(0, Texture)
```

If your application changed either of these settings since the state block was originally recorded, the **CaptureStateBlock** method will update the states within the block—and only those states—to their new values.

Applying State Blocks

[C++]
The **IDirect3DDevice8::ApplyStateBlock** method applies an existing state block to the device. The **ApplyStateBlock** method accepts a single parameter, *Token*, that identifies the state block to apply.

Note

You cannot apply a state block while recording another state block—that is, between calls to **IDirect3DDevice8::BeginStateBlock** and **IDirect3DDevice8::EndStateBlock**. Attempts to do so will fail.

[Visual Basic]
The **Direct3DDevice8.ApplyStateBlock** method applies an existing state block to the device. The **ApplyStateBlock** method accepts a single parameter, *Token*, that identifies the state block to apply.

Note

You cannot apply a state block while recording another state block—that is, between calls to **Direct3DDevice8.BeginStateBlock** and **Direct3DDevice8.EndStateBlock**. Attempts to do so will fail.

Creating Predefined State Blocks

[C++]
The **IDirect3DDevice8::CreateStateBlock** method creates a new state block that contains the entire set of device states or only those device states related to vertex or pixel processing. The **CreateStateBlock** method accepts two parameters. The first parameter identifies the type of state information to capture in the new state block, and the second parameter is the address of a variable that will receive a valid state-block handle if the call succeeds.
Valid values for the first parameter are defined by the **D3DSTATEBLOCKTYPE** enumerated type, which includes members that you can use to capture the entire set of device states (D3DSBT_ALL), or only those states that pertain to vertex or pixel processing (D3DSBT_VERTEXSTATE or D3DSBT_PIXELSTATE). The following list summarizes the states that the system captures when you pass the D3DSBT_VERTEXSTATE or D3DSBT_PIXELSTATE values.

- **Vertex-related states (D3DSBT_VERTEXSTATE)**
- **State (enabled or disabled) of all lights**
- **Transformation matrices**
- **User-defined clipping planes**

These values are set for the following render states.

D3DRS_AMBIENT
D3DRS_AMBIENTMATERIALSOURCE
D3DRS_CLIPPING
D3DRS_CLIPPLANEENABLE
D3DRS_COLORVERTEX
D3DRS_CULLMODE
D3DRS_DIFFUSEMATERIALSOURCE
D3DRS_EMISSIVEMATERIALSOURCE
D3DRS_FOGCOLOR
D3DRS_FOGDENSITY
D3DRS_FOGENABLE

D3DRS_FOGEND
D3DRS_FOGSTART
D3DRS_FOGTABLEMODE
D3DRS_FOGVERTEXMODE
D3DRS_INDEXEDVERTEXBLENDENABLE
D3DRS_LIGHTING
D3DRS_LOCALVIEWER
D3DRS_NORMALIZENORMALS
D3DRS_POINTSCALE_A
D3DRS_POINTSCALE_B
D3DRS_POINTSCALE_C
D3DRS_POINTSIZE
D3DRS_POINTSIZE_MAX
D3DRS_POINTSIZE_MIN
D3DRS_POINTSCALEENABLE
D3DRS_POINTSPRITEENABLE
D3DRS_PATCHEDGESTYLE
D3DRS_PATCHSEGMENTS
D3DRS_RANGEFOGENABLE
D3DRS_SHADEMODE
D3DRS_SOFTWAREVERTEXPROCESSING
D3DRS_SPECULARENABLE
D3DRS_SPECULARMATERIALSOURCE
D3DRS_TWEENFACTOR
D3DRS_VERTEXBLEND

These values are set for the following texture-stage states.

D3DTSS_TEXCOORDINDEX
D3DTSS_TEXTURETRANSFORMFLAGS

- **Pixel-related states (D3DSBT_PIXELSTATE)**

These values are set for the following render states.

D3DRS_ALPHABLENDENABLE
D3DRS_ALPHAFUNC
D3DRS_ALPHAREF
D3DRS_ALPHATESTENABLE
D3DRS_COLORWRITEENABLE
D3DRS_DESTBLEND
D3DRS_DITHERENABLE
D3DRS_EDGEANTIALIAS

D3DRS_FILLMODE
D3DRS_FOGDENSITY
D3DRS_FOGEND
D3DRS_FOGSTART
D3DRS_LASTPIXEL
D3DRS_LINEPATTERN
D3DRS_MULTISAMPLEANTIALIAS
D3DRS_MULTISAMPLEMASK
D3DRS_SHADEMODE
D3DRS_SRCBLEND
D3DRS_STENCILENABLE
D3DRS_STENCILFAIL
D3DRS_STENCILFUNC
D3DRS_STENCILMASK
D3DRS_STENCILPASS
D3DRS_STENCILREF
D3DRS_STENCILWRITEMASK
D3DRS_STENCILZFAIL
D3DRS_TEXTUREFACTOR
D3DRS_WRAP0 through D3DRS_WRAP7
D3DRS_ZBIAS
D3DRS_ZENABLE
D3DRS_ZFUNC
D3DRS_ZWRITEENABLE

These values are set for the following texture-stage states.

D3DTSS_ADDRESSU
D3DTSS_ADDRESSV
D3DTSS_ADDRESSW
D3DTSS_ALPHAARG1
D3DTSS_ALPHAARG2
D3DTSS_ALPHAOP
D3DTSS_BORDERCOLOR
D3DTSS_BUMPENVLOFFSET
D3DTSS_BUMPENVLSCALE
D3DTSS_BUMPENVMAT00
D3DTSS_BUMPENVMAT01
D3DTSS_BUMPENVMAT10
D3DTSS_BUMPENVMAT11

D3DTSS_COLORARG1
D3DTSS_COLORARG2
D3DTSS_COLOROP
D3DTSS_MAGFILTER
D3DTSS_MAXANISOTROPY
D3DTSS_MAXMIPLEVEL
D3DTSS_MINFILTER
D3DTSS_MIPFILTER
D3DTSS_MIPMAPLODBIAS
D3DTSS_TEXCOORDINDEX
D3DTSS_TEXTURETRANSFORMFLAGS

[Visual Basic]
The **Direct3DDevice8.CreateStateBlock** method creates a new state block that contains the entire set of device states or only those device states related to vertex or pixel processing. The **CreateStateBlock** method accepts two parameters. The first parameter identifies the type of state information to capture in the new state block, and the second parameter is the address of a variable that will receive a valid state-block handle if the call succeeds.
Valid values for the first parameter are defined by the **CONST_D3DSTATEBLOCKTYPE** enumeration, which includes members that you can use to capture the entire set of device states (D3DSBT_ALL), or only those states that pertain to vertex or pixel processing (D3DSBT_VERTEXSTATE or D3DSBT_PIXELSTATE). The following list summarizes the states that the system captures when you pass the D3DSBT_VERTEXSTATE or D3DSBT_PIXELSTATE values.

- **Vertex-related states (D3DSBT_VERTEXSTATE)**
- **State (enabled or disabled) of all lights**
- **Transformation matrices**
- **User-defined clipping planes**

These values are set for the following render states.
D3DRS_AMBIENT
D3DRS_AMBIENTMATERIALSOURCE
D3DRS_CLIPPING
D3DRS_CLIPPLANEENABLE
D3DRS_COLORVERTEX
D3DRS_CULLMODE
D3DRS_DIFFUSEMATERIALSOURCE
D3DRS_EMISSIVEMATERIALSOURCE

D3DRS_FOGCOLOR
D3DRS_FOGDENSITY
D3DRS_FOGENABLE
D3DRS_FOGEND
D3DRS_FOGSTART
D3DRS_FOGTABLEMODE
D3DRS_FOGVERTEXMODE
D3DRS_INDEXVERTEXBLENDENABLE
D3DRS_LIGHTING
D3DRS_LOCALVIEWER
D3DRS_NORMALIZENORMALS
D3DRS_POINTSCALE_A
D3DRS_POINTSCALE_B
D3DRS_POINTSCALE_C
D3DRS_POINTSIZE
D3DRS_POINTSIZE_MAX
D3DRS_POINTSIZE_MIN
D3DRS_POINTSCALEENABLE
D3DRS_POINTSPRITEENABLE
D3DRS_PATCHSEGMENTS
D3DRS_RANGEFOGENABLE
D3DRS_SHADEMODE
D3DRS_SOFTWAREVERTEXPROCESSING
D3DRS_SPECULARENABLE
D3DRS_SPECULARMATERIALSOURCE
D3DRS_TWEENFACTOR
D3DRS_VERTEXBLEND

These values are set for the following texture-stage states.

D3DTSS_TEXCOORDINDEX
D3DTSS_TEXTURETRANSFORMFLAGS

- **Pixel-related States (D3DSBT_PIXELSTATE)**

These values are set for the following render states.

D3DRS_ALPHABLENDENABLE
D3DRS_ALPHAFUNC
D3DRS_ALPHAREF
D3DRS_ALPHATESTENABLE
D3DRS_COLORWRITEENABLE
D3DRS_DESTBLEND

D3DRS_DITHERENABLE
D3DRS_EDGEANTIALIAS
D3DRS_FILLMODE
D3DRS_FOGDENSITY
D3DRS_FOGEND
D3DRS_FOGSTART
D3DRS_LASTPIXEL
D3DRS_LINEPATTERN
D3DRS_MULTISAMPLEANTIALIAS
D3DRS_MULTISAMPLEMASK
D3DRS_SHADEMODE
D3DRS_SRCBLEND
D3DRS_STENCILENABLE
D3DRS_STENCILFAIL
D3DRS_STENCILFUNC
D3DRS_STENCILMASK
D3DRS_STENCILPASS
D3DRS_STENCILREF
D3DRS_STENCILWRITEMASK
D3DRS_STENCILZFAIL
D3DRS_TEXTUREFACTOR
D3DRS_WRAP0 through D3DRS_WRAP7
D3DRS_ZBIAS
D3DRS_ZENABLE
D3DRS_ZFUNC
D3DRS_ZWRITEENABLE

These values are set for the following texture-stage states.

D3DTSS_ADDRESSU
D3DTSS_ADDRESSV
D3DTSS_ADDRESSW
D3DTSS_ALPHAARG1
D3DTSS_ALPHAARG2
D3DTSS_ALPHAOP
D3DTSS_BORDERCOLOR
D3DTSS_BUMPENVLOFFSET
D3DTSS_BUMPENVLSCALE
D3DTSS_BUMPENVMAT00
D3DTSS_BUMPENVMAT01

D3DTSS_BUMPENVMAT10
D3DTSS_BUMPENVMAT11
D3DTSS_COLORARG1
D3DTSS_COLORARG2
D3DTSS_COLOROP
D3DTSS_MAGFILTER
D3DTSS_MAXANISOTROPY
D3DTSS_MAXMIPLEVEL
D3DTSS_MINFILTER
D3DTSS_MIPFILTER
D3DTSS_MIPMAPLODBIAS
D3DTSS_TEXCOORDINDEX
D3DTSS_TEXTURETRANSFORMFLAGS

Note

It is important to check the error code from the **CreateStateBlock** method. If the method fails, it is likely because the display mode has changed. Your application should recover from this type of failure by recreating its surfaces, and then recreating the state block.

Deleting State Blocks

[C++]
The **IDirect3DDevice8::DeleteStateBlock** deletes a state block, deallocating the memory used to contain it. The **DeleteStateBlock** method accepts the handle to the state block to delete at its only parameter. After deleting a state block, any handles to it are invalid, and can no longer be used.

[Visual Basic]
The **Direct3DDevice8.DeleteStateBlock** deletes a state block, deallocating the memory used to contain it. The **DeleteStateBlock** method accepts the handle to the state block to delete at its only parameter. After deleting a state block, any handles to it are invalid, and can no longer be used.

Lost Devices

This section provides information about lost devices in Microsoft® Direct3D®. Information is divided into the following topics.

- What is a Lost Device?

- Responding to a Lost Device
- Lost Devices and Locking Operations
- Lost Devices and Resources
- Lost Devices and Retrieved Data

What is a Lost Device?

[C++]
A Microsoft® Direct3D® device can be in either an operational state or a lost state. The operational state is the normal state of the device; the device executes and presents all rendering as expected. The device makes a transition to the lost state when an event—such as the loss of keyboard focus in a full-screen application—causes rendering to become impossible. The lost state is characterized by the silent failure of all rendering operations, which means that the rendering methods can return success codes even though the rendering operations fail. In this situation, the error code D3DERR_DEVICELOST is returned by **IDirect3DDevice8::Present**.
By design, the full set of scenarios that can cause a device to become lost is not specified. Some typical examples include loss of focus, such as when the user presses ALT+TAB or when a system dialog is initialized. Devices can also be lost due to a power management event, or when another application assumes full-screen operation. In addition, any failure from **IDirect3DDevice8::Reset** puts the device into a lost state.

[Visual Basic]
A Microsoft® Direct3D® device can be in either an operational state or a lost state. The operational state is the normal state of the device; the device executes and presents all rendering as expected. The device makes a transition to the lost state when an event—such as the loss of keyboard focus in a full-screen application—causes rendering to become impossible. The lost state is characterized by the silent failure of all rendering operations, which means that the rendering methods can return success codes even though the rendering operations fail. In this situation, the error code D3DERR_DEVICELOST is returned by **Direct3DDevice8.Present**.
By design, the full set of scenarios that can cause a device to become lost is not specified. Some typical examples include loss of focus, such as when the user presses ALT+TAB or when a system dialog is initialized. Devices can also be lost due to a power management event, or when another application assumes full-screen operation. In addition, any failure from **Direct3DDevice8.Reset** puts the device into a lost state.

Responding to a Lost Device

[C++]
If a device is lost, the application queries the device to see if it can be restored to the operational state. If not, the application waits until the device can be restored.

If the device can be restored, the application prepares the device by destroying all video-memory resources and any swap chains. Then, the application calls the **IDirect3DDevice8::Reset** method. **Reset** is the only method that has an effect when a device is lost, and is the only method by which an application can change the device from a lost to an operational state. **Reset** will fail unless the application releases all resources that are allocated in D3DPOOL_DEFAULT, including those created by the **IDirect3DDevice8::CreateRenderTarget** and **IDirect3DDevice8::CreateDepthStencilSurface** methods.
For the most part, the high-frequency calls of Microsoft® Direct3D® do not return any information about whether the device has been lost. The application can continue to call rendering methods, such as **IDirect3DDevice8::DrawPrimitive**, without receiving notification of a lost device. Internally, these operations are discarded until the device is reset to the operational state.
The application can determine what to do on encountering a lost device by querying the return value of the **IDirect3DDevice8::TestCooperativeLevel** method.

- If the method returns D3D_OK, the device is operational.
- If the device is lost but cannot be restored at the current time, the return value is D3DERR_DEVICELOST. This is the case when the user presses ALT+TAB, causing a full-screen device to lose focus. Applications should respond by pausing until the device can be reset. A D3DERR_DEVICENOTRESET return code from **TestCooperativeLevel** indicates this situation.
- If the device is lost and can be restored, the return code from **TestCooperativeLevel** is D3DERR_DEVICENOTRESET. Note that the return code from the **IDirect3DDevice8::Present** method is still D3DERR_DEVICELOST.

In all cases, destroying video-memory resources is a prerequisite to calling **Reset**, even if the device has not been lost.

[Visual Basic]
If a device is lost, the application queries the device to see if it can be restored to the operational state. If not, the application waits until the device can be restored.
If the device can be restored, the application prepares the device by destroying all video-memory resources and any swap chains. Then, the application calls **Direct3DDevice8.Reset**. **Reset** is the only method that has an effect when the device is lost, and is the only method by which an application can change the device from a lost to an operational state. **Reset** will fail unless the application releases all resources that are allocated in D3DPOOL_DEFAULT, including those created by the **Direct3DDevice8.CreateRenderTarget** and **Direct3DDevice8.CreateDepthStencilSurface** methods.
For the most part, the high-frequency calls of Microsoft® Direct3D® do not return information about whether the device has been lost. The application can continue to call rendering methods, such as **Direct3DDevice8.DrawPrimitive**, without receiving notification of a lost device. Internally, these operations are discarded until the device is reset to the operational state.

The application can determine what to do on encountering a lost device by querying the return value of the **Direct3DDevice8.TestCooperativeLevel** method.

- If the method returns D3D_OK, the device is operational.
- If the device is lost but cannot be restored at the current time, the return value is D3DERR_DEVICELOST. This is the case when the user presses ALT+TAB, causing a full-screen device to lose focus. Applications should respond by pausing until the device can be reset. A D3DERR_DEVICENOTRESET return code from **TestCooperativeLevel** indicates this situation.
- If the device is lost and can be restored, the return code from **TestCooperativeLevel** is D3DERR_DEVICENOTRESET. Note that the return code from the **Direct3DDevice8.Present** method is still D3DERR_DEVICELOST.

In all cases, destroying video-memory resources is a prerequisite to calling **Reset**, even if the device has not been lost.

Lost Devices and Locking Operations

Internally, Microsoft® Direct3D® does enough work to ensure that a lock operation will succeed after a device is lost. However, it is not guaranteed that the video-memory resource's data will be accurate during the lock operation. It is guaranteed that no error code will be returned. This allows applications to be written without concern for device loss during a lock operation.
For more information, see Lost Devices and Retrieved Data.

Lost Devices and Resources

[C++]
Resources can consume video memory. Because a lost device is disconnected from the video memory owned by the adapter, it is not possible to guarantee allocation of video memory when the device is lost. As a result, all resource creation methods are implemented to succeed by returning D3D_OK, but do in fact allocate only dummy system memory. Because any video-memory resource must be destroyed before the device is resized, there is no issue of over-allocating video memory. These dummy surfaces allow lock and copy operations to appear to function normally until the application calls **IDirect3DDevice8::Present** and discovers that the device has been lost.
All video memory must be released before a device can be reset from a lost state to an operational state. This means that the application should release any swap chains created with **IDirect3DDevice8::CreateAdditionalSwapChain** and any resources placed in the D3DPOOL_DEFAULT memory class. The application need not release resources in the D3DPOOL_MANAGED or D3DPOOL_SYSTEMMEM memory classes. Other state data is automatically destroyed by the transition to an operational state.

[Visual Basic]
Resources can consume video memory. Because a lost device is disconnected from the video memory owned by the adapter, it is not possible to guarantee allocation of video memory when the device is lost. As a result, all resource creation methods are implemented to succeed by returning D3D_OK, but do in fact allocate only dummy system memory. Because any video-memory resource must be destroyed before the device is resized, there is no issue of over-allocating video memory. These dummy surfaces allow lock and copy operations to appear to function normally until the application calls **Direct3DDevice8.Present** and discovers that the device has been lost.
All video memory must be released before a device can be reset from a lost state to an operational state. This means that the application should release any swap chains created with **Direct3DDevice8.CreateAdditionalSwapChain** and any resources placed in the D3DPOOL_DEFAULT memory class. The application need not release resources in the D3DPOOL_MANAGED or D3DPOOL_SYSTEMMEM memory classes. Other state data is automatically destroyed by the transition to an operational state.

You are encouraged to develop applications with a single code path to respond to device loss. This code path is likely to be similar, if not identical, to the code path taken to initialize the device at startup.

Lost Devices and Retrieved Data

Microsoft® Direct3D® allows applications to copy generated or previously written images from video-memory resources to nonvolatile system-memory resources. Because the source images of such transfers might be lost at any time, Direct3D allows such copy operations to fail when the device is lost.

[C++]
The copy operations, **IDirect3DDevice8::UpdateTexture** and **IDirect3DDevice8::CopyRects**, can return D3DERR_DEVICELOST when the source object is in volatile memory (D3DPOOL_DEFAULT) and the destination object is in nonvolatile memory (D3DPOOL_SYSTEMMEM or D3DPOOL_MANAGED). Another copy operation, **IDirect3DDevice8::GetFrontBuffer**, can fail due to retrieving data from the primary surface. Note that these cases are the only instance of D3DERR_DEVICELOST outside of the **IDirect3DDevice8::Present**, **IDirect3DDevice8::TestCooperativeLevel**, and **IDirect3DDevice8::Reset** methods.

[Visual Basic]
The copy operations, **Direct3DDevice8.UpdateTexture** and **Direct3DDevice8.CopyRects**, can return D3DERR_DEVICELOST when the source object is in volatile memory (D3DPOOL_DEFAULT) and the destination object is in nonvolatile memory (D3DPOOL_SYSTEMMEM or D3DPOOL_MANAGED). Another copy operation, **Direct3DDevice8.GetFrontBuffer**, can fail due to retrieving data from the primary surface. Note that these cases are the only instance of

D3DERR_DEVICELOST outside of the **Direct3DDevice8.Present**, **Direct3DDevice8.TestCooperativeLevel**, and **Direct3DDevice8.Reset** methods.

Direct3D Resources

This section discusses resources and gives a general overview on how they are used in your application. Information is divided into the following topics.

- What Are Resources?
- Resource Properties
- Resource Relationships
- Using Resources

What Are Resources?

Resources are the textures and buffers that are used to render a scene. Applications need to create, load, copy, and use resources. This section gives a brief introduction to resources and the steps and methods used by applications when working with resources. For more information on specific resource types, see Textures, Vertex Buffers, and Index Buffers.

[C++]
All resources, including the geometry resources **IDirect3DIndexBuffer8** and **IDirect3DVertexBuffer8**, inherit from the **IDirect3DResource8** interface. The texture resources, **IDirect3DCubeTexture8**, **IDirect3DTexture8**, and **IDirect3DVolumeTexture8**, also inherit from the **IDirect3DBaseTexture8** interface.

[Visual Basic]
All resources, including the geometry resources **Direct3DIndexBuffer8** and **Direct3DVertexBuffer8**, implement the methods of the **Direct3DResource8** class. The texture resources, **Direct3DCubeTexture8**, **Direct3DTexture8**, and **Direct3DVolumeTexture8**, also implement the methods of the **Direct3DBaseTexture8** class.

Resource Properties

All resources share the following properties.

- *Usage*. The way a resource is used—for example, as a texture or a render target.
- *Format*. The format of the data—for example, the pixel format of a 2-D surface.
- *Pool*. The type of memory where the resource is allocated.

- *Type*. The type of resource—for example, a vertex buffer or render target.

Resource uses are enforced. An application that will use a resource in a certain operation must specify that operation at resource creation time. The following usages are defined for resources.

- D3DUSAGE_DEPTHSTENCIL
- D3DUSAGE_DONOTCLIP
- D3DUSAGE_DYNAMIC
- D3DUSAGE_RTPATCHES
- D3DUSAGE_NPATCHES
- D3DUSAGE_POINTS
- D3DUSAGE_RENDERTARGET
- D3DUSAGE_SOFTWAREPROCESSING
- D3DUSAGE_WRITEONLY

The D3DUSAGE_RTPATCHES, D3DUSAGE_NPATCHES, and D3DUSAGE_POINTS flags indicate to the driver that the data in these buffers is likely to be used for triangular or grid patches, N patches, or point sprites, respectively. These flags are provided in case the hardware cannot perform these operations without host processing. Therefore, the driver will want to allocate these surfaces in system memory so that the CPU can access them. If the driver can perform these operations entirely in hardware, then it can allocate these surfaces in video or /AGP memory to avoid a host copy and improve performance at least twofold. Note that the information provided by these flags is not absolutely required. A driver can detect that such operations are being performed on the data, and it will move the buffer back to system memory for subsequent frames.
For details on the usage flags and how they relate to specific resources, see the reference pages on the individual resource creation methods.

[C++]
For information on the surface format of resources, see the **D3DFORMAT** enumerated type.
The class of memory that holds a resource's buffers is called a pool. Pool values are defined by the **D3DPOOL** enumerated type. A pool cannot be mixed for different objects contained in a single resource—that is, mip levels in a mipmap—and once a pool is chosen for a resource, the pool cannot be changed.
The resources types are set implicitly at run time when the application calls a resource creation method such as **IDirect3DDevice8::CreateCubeTexture**. Resource types are defined by the **D3DRESOURCETYPE** enumerated type. Applications can query these types at run time; however, it is expected that most scenarios will not require run-time type checking.

[Visual Basic]
For information on the surface format of resources, see the **CONST_D3DFORMAT** enumeration.

The class of memory that holds a resource's buffers is called a pool. Pool values are defined by the **CONST_D3DPOOL** enumeration. A pool cannot be mixed for different objects contained in a single resource—that is, mip levels in a mipmap—and once a pool is chosen for a resource, the pool cannot be changed.
The resources types are set implicitly at run time when the application calls a resource creation method such as **Direct3DDevice8.CreateCubeTexture**. Resource types are defined by the **CONST_D3DRESOURCETYPE** enumeration. Applications can query these types at run time; however, it is expected that most scenarios will not require run-time type checking.

Resource Relationships

The following diagram illustrates the operations that are syntactically possible on specific resources and their contents. For more information on how to use resources, see Manipulating Resources.

Arrows running from left to right indicate that the target types are created by the given methods, while arrows running from right to left indicate that those resource types can be passed as arguments to the given methods.

Using Resources

The following topics discuss common tasks that applications perform when working with resources.

- Managing Resources
- Application-Managed Resources and Allocation Strategies
- Manipulating Resources
- Locking Resources

Managing Resources

[C++]
Resource management is the process where resources are promoted from system-memory storage to device-accessible storage and discarded from device-accessible storage. The Microsoft® Direct3D® run time has its own management algorithm based on a least-recently-used priority technique. Direct3D switches to a most-recently-used priority technique when it detects that more resources than can coexist in device-accessible memory are used in a single frame—between **IDirect3DDevice8::BeginScene** and **IDirect3DDevice8::EndScene** calls.
Use the D3DPOOL_MANAGED flag at creation time to specify a managed resource. Managed resources persist through transitions between the lost and operational states of the device. The device can be restored with a call to **IDirect3DDevice8::Reset**,

and such resources continue to function normally without being reloaded with artwork. However, if the device must be destroyed and recreated, all resources created using D3DPOOL_MANAGED must be recreated.
Use the D3DPOOL_DEFAULT flag at creation time to specify that a resource be placed in the default pool. Resources in the default pool do not persist through transitions between the lost and operational states of the device. These resources must be released before calling **Reset** and must then be recreated.

[Visual Basic]
Resource management is the process where resources are promoted from system-memory storage to device-accessible storage and discarded from device-accessible storage. The Microsoft® Direct3D® run time has its own management algorithm based on a least-recently-used priority technique. Direct3D switches to a most-recently-used priority technique when it detects that more resources than can coexist in device-accessible memory are used within a single frame—between **Direct3DDevice8.BeginScene** and **Direct3DDevice8.EndScene** calls.
Use the D3DPOOL_MANAGED flag at creation time to specify a managed resource. Managed resources persist through transitions between the lost and operational states of the device. The device can be restored with a call to **Direct3DDevice8.Reset**, and such resources continue to function normally without being reloaded with artwork. However, if the device must be destroyed and recreated, all resources created using D3DPOOL_MANAGED must be recreated.
Use the D3DPOOL_DEFAULT flag at creation time to specify that a resource be placed in the default pool. Resources in the default pool do not persist through transitions between the lost and operational states of the device. These resources must be released before calling **Reset** and must then be recreated.

For more information on the lost state of a device, see Lost Devices.
Note that resource management is not supported for all types and usages. For example, objects created with the D3DUSAGE_RENDERTARGET flag are not supported. In addition, resource management is not recommended for objects whose contents are changing with high frequency. For example, a managed vertex buffer that changes every frame can significantly degrade performance for some hardware. However, this is not a problem for texture resources.

Application-Managed Resources and Allocation Strategies

Managed vertex-buffer or index-buffer resources cannot be declared dynamic by specifying D3DUSAGE_DYNAMIC at creation time. This would require an additional copy for every modification to the vertex buffer contents. Dynamic vertex buffers are intended for rendering dynamic geometry as well as data pulled in from BSP trees or other visibility data structures. This can be accomplished by pre-allocating buffers of the desired format. These resources are then parceled out to support application needs by a resource manager within the application. Because there are only a few different vertex strides that an application will use simultaneously, and a different vertex buffer is required only for each unique stride,

the total number of dynamic vertex buffers is small. When managing dynamic resources in this way, it is important to ensure that high-frequency demands on the resources do not significantly decrease the application's performance.
When simultaneously using both D3D-managed and application-managed resources, all application managed resources should be allocated in D3DPOOL_DEFAULT memory prior to creating any managed resources. Allocating resources in D3DPOOL_DEFAULT memory after allocating resources in D3DPOOL_MANAGED memory will give the D3D memory manager an inaccurate account of available memory.

Manipulating Resources

[C++]
Your application manipulates resources in order to render a scene. First, the application creates texture resources by using the following methods.

- **IDirect3DDevice8::CreateCubeTexture**
- **IDirect3DDevice8::CreateTexture**
- **IDirect3DDevice8::CreateVolumeTexture**

The texture objects returned by the texture creation methods are containers for surfaces or volumes; these containers are generically known as buffers. The buffers owned by the resource inherit the usages, format, and pool of the resource but have their own type. For more information, see Resource Properties.
The application gains access to the contained surfaces, for the purpose of loading artwork, by calling the following methods. For details, see Locking Resources.

- **IDirect3DCubeTexture8::LockRect**
- **IDirect3DTexture8::LockRect**
- **IDirect3DVolumeTexture8::LockBox**

The lock methods take arguments denoting the contained surface—for example, the mipmap sub-level or cube face of the texture—and return pointers to the pixels. The typical application never uses a surface object directly.
In addition, the application creates geometry-oriented resources by using the following methods.

- **IDirect3DDevice8::CreateIndexBuffer**
- **IDirect3DDevice8::CreateVertexBuffer**

Your application locks and fills the buffer resources by calling the following methods.

- **IDirect3DIndexBuffer8::Lock**
- **IDirect3DVertexBuffer8::Lock**

If your application is allowing the Microsoft® Direct3D® run time to manage these resources, then the resource creation process ends here. Otherwise, the application manages the promotion of system memory resources to device-accessible resources, where the hardware accelerator can use them, by calling the **IDirect3DDevice8::UpdateTexture** method.

To present images rendered from resources, the application also needs color and depth-stencil buffers. For typical applications, the color buffer is owned by the device's swap chain, which is a collection of back buffer surfaces, and is implicitly created with the device. Depth-stencil surfaces can be implicitly created, or explicitly created by using the **IDirect3DDevice8::CreateDepthStencilSurface** method. The application associates a device and its depth and color buffer with a call to **IDirect3DDevice8::SetRenderTarget**.
For details on presenting the final image, see Presenting a Scene.

[Visual Basic]
Your application manipulates resources in order to render a scene. First, the application creates texture resources by using the following methods.

- **Direct3DDevice8.CreateCubeTexture**
- **Direct3DDevice8.CreateTexture**
- **Direct3DDevice8.CreateVolumeTexture**

The texture objects returned by the texture creation methods are containers for surfaces or volumes; these containers are generically known as buffers. The buffers owned by the resource inherit the usages, format, and pool of the resource but have their own type. For more information, see Resource Properties.
The application gains access to the contained surfaces, for the purpose of loading artwork, by calling the following methods. For details, see Locking Resources.

- **Direct3DCubeTexture8.LockRect**
- **Direct3DTexture8.LockRect**
- **Direct3DVolumeTexture8.LockBox**

The lock methods take arguments denoting the contained surface—for example, the mipmap sub-level or cube face of the texture—and return pointers to the pixels. The typical application never uses a surface object directly.
In addition, the application creates geometry-oriented resources by using the following methods.

- **Direct3DDevice8.CreateIndexBuffer**
- **Direct3DDevice8.CreateVertexBuffer**

Your application locks and fills the buffer resources by calling the following functions.

- **D3DIndexBuffer8SetData**
- **D3DVertexBuffer8SetData**

If your application is allowing the Microsoft® Direct3D® run time to manage these resources, then the resource creation process ends here. Otherwise, the application manages the promotion of system memory resources to device-accessible resources, where the hardware accelerator can use them, by calling **Direct3DDevice8.UpdateTexture**.

To present images rendered from resources, the application also needs color and depth-stencil buffers. For typical applications, the color buffer is owned by the device's swap chain, which is a collection of back buffer surfaces, and is implicitly created with the device. Depth-stencil surfaces can be implicitly created, or explicitly created by using the **Direct3DDevice8.CreateDepthStencilSurface** method. The application associates a device and its depth and color buffer with a call to **Direct3DDevice8.SetRenderTarget**.
For details on presenting the final image, see Presenting a Scene.

Locking Resources

Locking a resource means granting CPU access to its storage. The following locking methods are defined for resources.

- D3DLOCK_DISCARD
- D3DLOCK_READONLY
- D3DLOCK_NOOVERWRITE
- D3DLOCK_NOSYSLOCK
- D3DLOCK_NO_DIRTY_UPDATE

For details on locking flags and how they relate to specific resources, see the reference pages on the individual resource locking methods. Application developers should note that the D3DLOCK_DISCARD, D3DLOCK_READONLY, and D3DLOCK_NOOVERWRITE flags are only hints. The run time does not check that applications are following the functionality specified by these flags. An application that specifies D3DLOCK_READONLY but then writes to the resource should expect undefined results. An application that specifies D3DUSAGE_WRITEONLY but then reads from the resource should expect a significant performance penalty. In general, working against locking flags, including the locking usage flags, is not guaranteed to work in the future and may incur a significant performance penalty both now and in the future.
A lock operation is followed by an unlock operation. For example, after locking a texture, the application subsequently relinquishes direct access to locked textures by unlocking them. In addition to granting processor access, any other operations involving that resource are serialized for the duration of a lock. Only a single lock for a resource is allowed, even for non-overlapping regions, and no accelerator operations on a surface can be ongoing while a lock operation is outstanding on that surface.
Each resource interface has methods for locking the contained buffers. Each texture resource can also lock a sub-portion of that resource. Two-dimensional resources (surfaces) allow the locking of sub-rectangles, and volume resources allow the locking of sub-volumes or boxes. Each lock method returns a structure that contains a pointer to the storage backing the resource and values representing the distances between rows or planes of data, depending on the resource type. For details, see the method lists for the resource interfaces. The returned pointer always points to the top-left byte in the locked sub-regions.

When working with index and vertex buffers, you can make multiple lock calls; however, you must ensure that the number of lock calls matches the number of unlock calls.
The DXT set of compressed texture formats, which store pixels in encoded 4×4 blocks, can only be locked on 4×4 boundaries.

Surfaces

This section contains information about Microsoft® Direct3D® surfaces. The following topics are discussed.

- Basic Concepts of Surfaces
- Gamma Controls
- Accessing Surface Memory Directly
- Private Surface Data

Basic Concepts of Surfaces

This section contains information about the basic concepts associated with Microsoft® Direct3D® surfaces. The following topics are discussed.

- Surface Interfaces
- Width vs. Pitch
- Surface Formats
- Flipping Surfaces
- Copying to Surfaces

Surface Interfaces

[C++]
A surface represents a linear area of display memory. A surface usually resides in the display memory of the display card, although surfaces can exist in system memory. Surface objects are contained within the **IDirect3DSurface8** interface.
An **IDirect3DSurface8** interface is obtained by calling one of the following methods.

- **IDirect3DCubeTexture8::GetCubeMapSurface**
- **IDirect3DDevice8::CreateDepthStencilSurface**
- **IDirect3DDevice8::CreateImageSurface**
- **IDirect3DDevice8::CreateRenderTarget**
- **IDirect3DDevice8::GetBackBuffer**
- **IDirect3DDevice8::GetDepthStencilSurface**
- **IDirect3DDevice8::GetFrontBuffer**
- **IDirect3DDevice8::GetRenderTarget**
- **IDirect3DSwapChain8::GetBackBuffer**
- **IDirect3DTexture8::GetSurfaceLevel**

The **IDirect3DSurface8** interface enables you to indirectly access memory through the **IDirect3DDevice8::CopyRects** method. This method allows you to copy a rectangular region of pixels from one **IDirect3DSurface8** interface to another **IDirect3DSurface8** interface. The surface interface also has methods to directly access display memory. For example, you can use the **IDirect3DSurface8::LockRect** method to lock a rectangular region of display memory. It is important to call **IDirect3DSurface8::UnlockRect** after you are done working with the locked rectangular region on the surface.

[Visual Basic]
A surface represents a linear area of display memory. A surface usually resides in the display memory of the display card, although surfaces can exist in system memory. Surface objects are contained in the **Direct3DSurface8** object.
A **Direct3DSurface8** class is obtained by calling one of the following methods.

- **Direct3DCubeTexture8.GetCubeMapSurface**
- **Direct3DDevice8.CreateDepthStencilSurface**
- **Direct3DDevice8.CreateImageSurface**
- **Direct3DDevice8.CreateRenderTarget**
- **Direct3DDevice8.GetBackBuffer**
- **Direct3DDevice8.GetDepthStencilSurface**
- **Direct3DDevice8.GetFrontBuffer**
- **Direct3DDevice8.GetRenderTarget**
- **Direct3DSwapChain8.GetBackBuffer**
- **Direct3DTexture8.GetSurfaceLevel**

The Direct3DSurface8 object enables you to indirectly access memory through the **Direct3DDevice8.CopyRects** method. This method allows you to copy a rectangular region of pixels from one **Direct3DSurface8** object to another specified rectangular region on another **Direct3DSurface8** object. The surface object also has methods to directly access display memory. For example, you can use the **Direct3DSurface8.LockRect** method to lock a rectangular region of display memory. It is important to call **Direct3DSurface8.UnlockRect** after you are done working with the locked rectangular region on the surface.

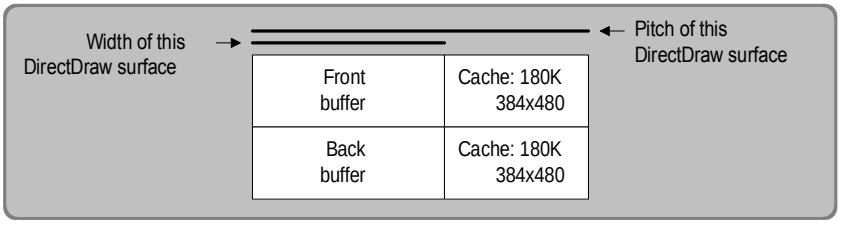
Width vs. Pitch

Although the terms *width* and *pitch* are often used informally, they have very important, and distinctly different, meanings. As a result, you should understand the meanings for each, and how to interpret the values that Microsoft® Direct3D® uses to describe them.

[C++]
Direct3D uses the **D3DSURFACE_DESC** structure to carry information describing a surface. Among other things, this structure is defined to contain information about a

surface's dimensions, as well as how those dimensions are represented in memory. The structure uses the **Height** and **Width** members to describe the logical dimensions of the surface. Both members are measured in pixels. Therefore, the **Height** and **Width** values for a 640×480 surface are the same whether it is an 8-bit surface or a 24-bit RGB surface.
When you lock a surface using the **IDirect3DSurface8::LockRect** method, the method fills in a D3DLOCKED_RECT structure that contains the pitch of the surface and a pointer to the locked bits. The value in the **Pitch** member describes the surface's memory pitch, also called *stride*. Pitch is the distance, in bytes, between two memory addresses that represent the beginning of one bitmap line and the beginning of the next bitmap line. Because pitch is measured in bytes rather than pixels, a 640×480×8 surface has a very different pitch value than a surface with the same dimensions but a different pixel format. Additionally, the pitch value sometimes reflects bytes that Direct3D has reserved as a cache, so it is not safe to assume that pitch is simply the width multiplied by the number of bytes per pixel. Rather, visualize the difference between width and pitch as shown in the following illustration.

[Visual Basic]
Direct3D uses the **D3DSURFACE_DESC** type to carry information describing a surface. Among other things, this structure is defined to contain information about a surface's dimensions, as well as how those dimensions are represented in memory. The structure uses the **Height** and **Width** members to describe the logical dimensions of the surface. Both members are measured in pixels. Therefore, the **Height** and **Width** values for a 640×480 surface are the same, whether it is an 8-bit surface or a 24-bit RGB surface.
When you lock a surface using the **Direct3DSurface8.LockRect** method, the method fills in a D3DLOCKED_RECT structure that contains the pitch of the surface and a pointer to the locked bits. The value in the **Pitch** member describes the surface's memory pitch, also called *stride*. Pitch is the distance, in bytes, between two memory addresses that represent the beginning of one bitmap line and the beginning of the next bitmap line. Because pitch is measured in bytes rather than pixels, a 640×480×8 surface has a very different pitch value than a surface with the same dimensions but a different pixel format. Additionally, the pitch value sometimes reflects bytes that Direct3D has reserved as a cache, so it is not safe to assume that pitch is simply the width multiplied by the number of bytes per pixel. Rather, visualize the difference between width and pitch as shown in the following illustration.



In this figure, the front buffer and back buffer are both 640×480×8, and the cache is 384×480×8.

[C++]
Pitch values are only useful when you are directly accessing surface memory. For example, after calling the **LockRect** method, the **pBits** member of the associated D3DLOCKED_RECT structure contains the address of the top-left pixel of the locked area of the surface, and the **Pitch** member is the surface pitch. You access pixels horizontally by incrementing or decrementing the surface pointer by the number of bytes per pixel, and you move up or down by adding the pitch value to, or subtracting it from, the current surface pointer.

[Visual Basic]
Pitch values are only useful when you are directly accessing surface memory. For example, after calling the **LockRect** method, the **pBits** member of the associated D3DLOCKED_RECT structure contains the address of the top-left pixel of the locked area of the surface, and the **Pitch** member is the surface pitch. You access pixels horizontally by incrementing or decrementing the surface pointer by the number of bytes per pixel, and you move up or down by adding the pitch value to, or subtracting it from, the current surface pointer.

When accessing surfaces directly, take care to stay within the memory allocated for the dimensions of the surface and stay out of any memory reserved for cache. Additionally, when you lock only a portion of a surface, you must stay within the rectangle you specify when locking the surface. Failing to follow these guidelines will have unpredictable results. When rendering directly into surface memory, always use the pitch returned by the **LockRect** method. Do not assume a pitch based solely on the display mode. If your application works on some display adapters but looks garbled on others, this may be the cause of the problem.
For more information, see Accessing Surface Memory Directly.

Surface Formats

[C++]
Surface formats dictate how data for each pixel in surface memory is interpreted. Microsoft® Direct3D® uses the **D3DFORMAT** member of the **D3DSURFACE_DESC** structure to describe the surface format. You can retrieve the format of an existing surface by calling the **IDirect3DSurface8::GetDesc** method.

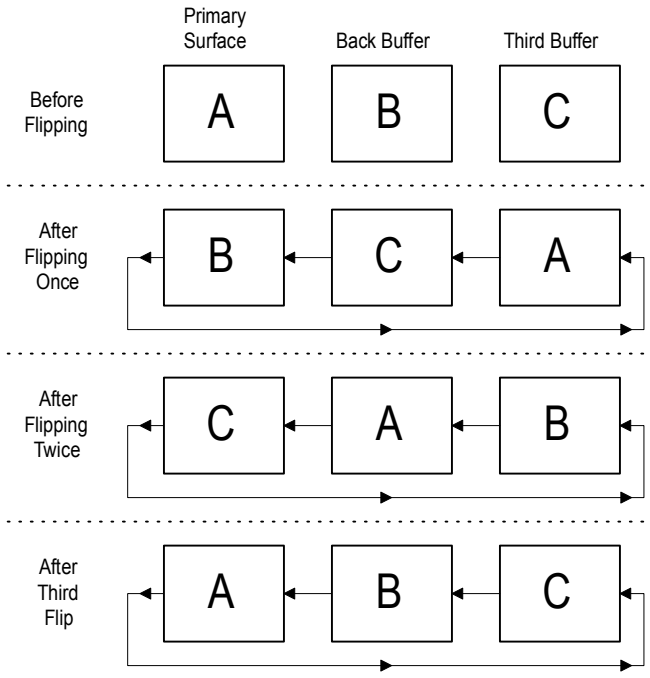
[Visual Basic]
Surface formats dictate how data for each pixel in surface memory is interpreted. Microsoft® Direct3D® uses the **CONST_D3DFORMAT** member of the **D3DSURFACE_DESC** structure to describe a surface format. You can retrieve the format of an existing surface by calling the **Direct3DSurface8.GetDesc** method.

Flipping Surfaces

A Microsoft® Direct3D® application typically displays an animated sequence by generating the frames of the animation in back buffers and presenting them in sequence. Back buffers are organized into swap chains. A swap chain is a series of buffers that "flip" to the screen one after another. This can be used to render one scene in memory and then flip the scene to the screen when rendering is complete. This avoids the phenomenon known as tearing and allows for smoother animation.

[C++]
Each device created in Direct3D has at least one swap chain. When you initialize the first Direct3D device, you set the *BackBufferCount* member of the **D3DPRESENT_PARAMETERS** structure, which tells Direct3D the number of back buffers that will be in the swap chain. The call to **IDirect3D8::CreateDevice** then creates the Direct3D device and corresponding swap chain.
When you use the **IDirect3DDevice8::Present** method to request a surface flip operation, the pointers to surface memory for the front buffer and back buffers are swapped. Flipping is performed by switching pointers that the display device uses for referencing memory, not by copying surface memory. When a flipping chain contains a front buffer and more than one back buffer, the pointers are switched in a circular pattern, as shown in the following illustration.

[Visual Basic]
Each device created in Direct3D has at least one swap chain. When you initialize the first Direct3D device, you set the *BackBufferCount* member of the **D3DPRESENT_PARAMETERS** structure, which tells Direct3D the number of back buffers that will be in the swap chain. The call to **Direct3D8.CreateDevice** then creates the Direct3D device and corresponding swap chain.
When you use the **Direct3DDevice8.Present** method to request a surface flip operation, the pointers to surface memory for the front buffer and back buffers are swapped. Flipping is performed by switching pointers that the display device uses for referencing memory, not by copying surface memory. When a flipping chain contains a front buffer and more than one back buffer, the pointers are switched in a circular pattern, as shown in the following illustration.



[C++]
You can create addition swap chains for a device by calling **IDirect3DDevice8::CreateAdditionalSwapChain**. An application can create one swap chain per view and associate each swap chain with a particular window. The application renders images in the back buffers of each swap chain, and then presents them individually. The two parameters that **CreateAdditionalSwapChain** takes are a pointer to a **D3DPRESENT_PARAMETERS** structure and the address of a pointer to an **IDirect3DSwapChain8** interface. You can then use **IDirect3DSwapChain8::Present** to display the contents of the next back buffer to the front buffer. Note that a device can only have one full-screen swap chain.
You can gain access to a specific back buffer by calling the **IDirect3DDevice8::GetBackBuffer** or **IDirect3DSwapChain8::GetBackBuffer** methods, which return a pointer to a **IDirect3DSurface8** interface that represents the returned back buffer surface. Note that calling this method increases the internal reference count on the **IDirect3DDevice8** interface so be sure to call **IUnknown::Release** when you are done using this surface or you will have a memory leak.
Remember, Direct3D flips surfaces by swapping surface memory pointers within the swap chain, not by swapping the surfaces themselves. This means that you will always render to the back buffer that will be displayed next.

[Visual Basic]

You can create addition swap chains for a device by calling **Direct3DDevice8.CreateAdditionalSwapChain**. An application can create one swap chain per view and associate each swap chain with a particular window. The application renders images in the back buffers of each swap chain, and then presents them individually. **CreateAdditionalSwapChain** takes a D3DPRESENT_PARAMETERS type and returns a **Direct3DSwapChain8** object. You can then use **Direct3DSwapChain8.Present** to display the contents of the next back buffer to the front buffer. Note that a device can only have one full-screen swap chain.
You can gain access to a specific back buffer by calling the **Direct3DDevice8.GetBackBuffer** or **Direct3DSwapChain8.GetBackBuffer** methods, which return a **Direct3DSurface8** object that represents the returned back buffer surface.
Remember, Direct3D flips surfaces by swapping surface memory pointers within the swap chain, not by swapping the surfaces themselves. This means that you will always render to the back buffer that is will be displayed next.

It is important to note the distinction between a "flipping operation", as performed by a display adapter driver, and a "Present" operation applied to a swap chain created with D3DSWAPEFFECT_FLIP.
The term "flip" conventionally denotes an operation that alters the range of video memory addresses that a display adapter uses to generate its output signal, thus causing the contents of a previously hidden back buffer to be displayed. In DirectX 8, the term is often used more generally to describe the presentation of a back buffer in any swap chain created with the D3DSWAPEFFECT_FLIP swap effect.
While such "Present" operations are almost invariably implemented by flip operations when the swap chain is a full-screen one, they are necessarily implemented by copy operations when the swap chain is windowed. Furthermore, a display adapter driver may use flipping to implement Present operations against full-screen swap chains based on the D3DSWAPEFFECT_DISCARD, D3DSWAPEFFECT_COPY and D3DSWAPEFFECT_COPY_VSYNC swap effects.
The discussion above applies to the commonly used case of a full-screen swap chain created with D3DSWAPEFFECT_FLIP.

[C++]
For a more general discussion of the different swap effects for both windowed and full-screen swap chains, see **D3DSWAPEFFECT**.

[Visual Basic]
For a more general discussion of the different swap effects for both windowed and full-screen swap chains, see **CONST_3DSWAPEFFECT**.

Copying To Surfaces

[C++]

When using the **IDirect3DDevice8::CopyRects** method, you pass an array of rectangles on the source surface or NULL to specify the entire surface. You also pass an array of points on the destination surface to which the top-left position of each rectangle on the source image is copied. This method does not support clipping. The operation will fail unless all the source rectangles and their corresponding destination rectangles are completely contained within the source and destination surfaces respectively. This method does not support alpha blending, color keys, or format conversion. Note that the destination and source surfaces must be distinct.

CopyRects Example

The following example copies two rectangles from the source surface to a destination surface. The first rectangle is copied from (0, 0, 50, 50) on the source surface to the same location on the destination surface, and the second rectangle is copied from (50, 50, 100, 100) on the source surface to (150 ,150, 200, 200) on the destination surface.

```
//The following assumptions are made:
//-d3dDevice is a valid Direct3DDevice8 object.
//-pSource and pDest are valid IDirect3DSurface8 pointers.

RECT  rcSource[] = {  0,  0,  50,  50,
                      50, 50, 100, 100 };
POINT ptDest[]   = {  0,  0, 150, 150 };

d3dDevice->CopyRect( pSource, rcSource, 2, pDest, ptDest);
```

The following methods are also available in C++/C for copying images to a Microsoft® Direct3D® surface.

- **D3DXLoadSurfaceFromFileA**
- **D3DXLoadSurfaceFromFileInMemory**
- **D3DXLoadSurfaceFromFileW**
- **D3DXLoadSurfaceFromMemory**
- **D3DXLoadSurfaceFromResourceA**
- **D3DXLoadSurfaceFromResourceW**
- **D3DXLoadSurfaceFromSurface**
- **IDirect3DDevice8::CopyRects**

[Visual Basic]

When using the **Direct3DDevice8.CopyRects** method, you pass the first element of an array of rectangles on the source surface or Nothing to specify the entire surface. You also pass an array of points on the destination surface to which the top-left-position of each rectangle on the source image is copied. You can pass ByVal 0 to copy the rectangles with the same top-left position as the source. This method does not support clipping. The operation will fail unless all the source rectangles and their corresponding destination rectangles are completely contained within the source and destination surfaces respectively. This method does not support alpha blending, color

keys, or format conversion. Note that the source and destination surfaces must be distinct.

CopyRects Example

The following example copies two rectangles from the source surface to a destination surface. The first rectangle is copied from (0, 0, 50, 50) on the source surface to the same location on the destination surface and the second rectangle is copied from (50, 50, 100, 100) on the source surface to (150 ,150, 200, 200) on the destination surface.

```
'The following assumptions are made:
'-d3dDevice is a valid IDirect3DDevice8 object.
'-Source and Dest are valid Direct3DSurface8 objects.

Dim rcSource(1) As RECT
Dim ptDest(1) As Point

rcSource(0).left = 0
rcSource(0).top = 0
rcSource(0).right= 0
rcSource(0).bottom = 0

rcSource(1).left = 50
rcSource(1).top = 50
rcSource(1).right = 100
rcSource(1).bottom = 100

ptDest(0).x = 0
ptDest(0).y = 0

ptDest(1).x = 150
ptDest(1).y = 150

d3dDevice.CopyRects Source, rcSource, 2, Dest, ptDest
```

The following methods are also available in Microsoft® Visual Basic® for copying images to a Microsoft Direct3D® surface.

- **D3DX8.LoadSurfaceFromFile**
- **D3DX8.LoadSurfaceFromFileInMemory**
- **D3DX8.LoadSurfaceFromMemory**
- **D3DX8.LoadSurfaceFromResource**
- **D3DX8.LoadSurfaceFromSurface**
- **Direct3DDevice8.CopyRects**

Gamma Controls

This section contains information about gamma controls used with Microsoft® Direct3D®. Information is organized into the following topics.

- What Are Gamma Controls?
- Using Gamma Controls

What Are Gamma Controls?

Gamma controls allow you to change how the system displays the contents of the surface, without affecting the contents of the surface itself. Think of these controls as very simple filters that Microsoft® Direct3D® applies to data as it leaves a surface and before it is rendered on the screen.

Gamma controls are simply a property of a swap chain. Gamma controls make it possible to dynamically change how a surface's red, green, and blue levels map to the actual levels that the system displays. By setting gamma levels, you can cause the user's screen to flash colors—red when the user's character is shot, green when the character picks up a new item, and so on—without copying new images to the frame buffer to achieve the effect. Or, you might adjust color levels to apply a color bias to the images in the back buffer.

There is always at least one swap chain (the implicit swap chain) for each device because Microsoft® Direct3D® for Microsoft DirectX® 8.0 has one swap chain as a property of the device. Because the gamma ramp is a property of the swap chain, the gamma ramp can be applied when the swap chain is windowed. The gamma ramp takes effect immediately. There is no waiting for a VSYNC operation.

For details on how to use gamma controls, see Using Gamma Controls.

Using Gamma Controls

[C++]
The **IDirect3DDevice8::SetGammaRamp** and **IDirect3DDevice8::GetGammaRamp** methods allow you to manipulate ramp levels that affect the red, green, and blue color components of pixels from the surface before they are sent to the digital-to-analog converter (DAC) for display.

[Visual Basic]
The **Direct3DDevice8.SetGammaRamp** and **Direct3DDevice8.GetGammaRamp** methods allow you to manipulate ramp levels that affect the red, green, and blue color components of pixels from the surface before they are sent to the digital-to-analog converter (DAC) for display.

In the following topics describe the general concept of ramp levels and provide information about working with those levels.

- About Gamma Ramp Levels
- Detecting Gamma Ramp Support

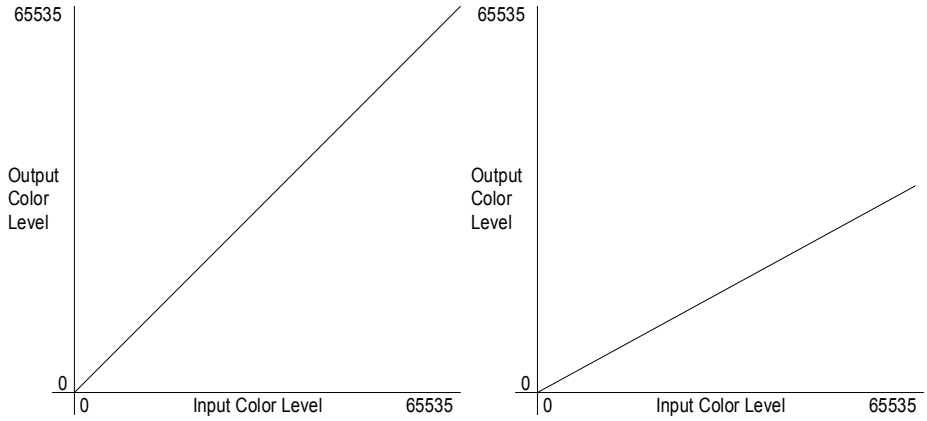
- Setting and Retrieving Gamma Ramp Levels

About Gamma Ramp Levels

In Microsoft® Direct3D®, the term *gamma ramp* describes a set of values that map the level of a particular color component—red, green, blue—for all pixels in the frame buffer to new levels that are received by the digital-to-analog converter (DAC) for display. The remapping is performed by way of three look-up tables, one for each color component.

Here's how it works: Direct3D takes a pixel from the frame buffer and evaluates its individual red, green, and blue color components. Each component is represented by a value from 0 to 65535. Direct3D takes the original value and uses it to index a 256-element array (the ramp), where each element contains a value that replaces the original one. Direct3D performs this look-up and replace process for each color component of each pixel in the frame buffer, thereby changing the final colors for all the on-screen pixels.

It's handy to visualize the ramp values by graphing them. The left graph of the two following graphs shows a ramp that doesn't modify colors at all. The right graph shows a ramp that imposes a negative bias to the color component to which it is applied.



The array elements for the graph on the left contain values identical to their index—0 in the element at index 0, and 65535 at index 255. This type of ramp is the default, as it doesn't change the input values before they're displayed. The right graph provides more variation; its ramp contains values that range from 0 in the first element to 32768 in the last element, with values ranging uniformly in between. The effect is that the color component that uses this ramp appears muted on the display. You are not limited to using linear graphs; if your application can assign arbitrary mapping if needed. You can even set the entries to all zeroes to remove a color component completely from the display.

Detecting Gamma Ramp Support

If the device does not support gamma ramps in the swap chain's current presentation mode (full-screen or windowed), no error value is returned. Applications can check the D3DCAPS2_FULLSCREENGAMMA and D3DCAPS2_CANCALIBRATEGAMMA capability bits in the Caps2 member of the D3DCAPS8 type to determine the capabilities of the device and whether a calibrator is installed.

Setting and Retrieving Gamma Ramp Levels

[C++]
Gamma ramp levels are effectively look-up tables that Microsoft® Direct3D® uses to map the frame buffer color components to new levels that will be displayed. For more information, see About Gamma Ramp Levels. You can set and retrieve ramp levels for the primary surface by calling the **IDirect3DDevice8::SetGammaRamp** and **IDirect3DDevice8::GetGammaRamp** methods. **SetGammaRamp** accepts two parameters and **GetGammaRamp** accepts one parameter. For **SetGammaRamp**, the first parameter is either D3DSGR_CALIBRATE or D3DSGR_NO_CALIBRATION. The second parameter, *pRamp*, is a pointer to a **D3DGAMMARAMP** structure. The **D3DGAMMARAMP** structure contains three 256-element arrays of **WORD**s, one array each to contain the red, green, and blue gamma ramps. **GetGammaRamp** has one parameter that takes a pointer to a **D3DGAMMARAMP** type that will be filled with the current gamma ramp.
You can include the DDSGR_CALIBRATE value for the first parameter of **SetGammaRamp** to invoke the calibrator when setting new gamma levels. Calibrating gamma ramps incurs some processing overhead, and should not be used frequently. Setting a calibrated gamma ramp provides a consistent and absolute gamma value for the user, regardless of the display adapter and monitor.
Not all systems support gamma calibration. To determine if gamma calibration is supported, call **IDirect3DDevice8::GetDeviceCaps**, and examine the **Caps2** member of the associated **D3DCAPS8** structure after the method returns. If the D3DCAPS2_CANCALIBRATEGAMMA capability flag is present, then gamma calibration is supported.
When setting new ramp levels, keep in mind that that the levels you set in the arrays are only used when your application is in full-screen, exclusive mode. Whenever your application changes to normal mode, the ramp levels are set aside, taking effect again when the application reinstates full-screen mode.

[Visual Basic]
Gamma ramp levels are effectively look-up tables that Microsoft® Direct3D® uses to map the frame buffer color components to new levels that will be displayed. For more information, see About Gamma Ramp Levels. You set and retrieve ramp levels for the primary surface by calling the **Direct3DDevice8.SetGammaRamp** and **Direct3DDevice8.GetGammaRamp** methods. **SetGammaRamp** accepts two parameters and **GetGammaRamp** accepts one parameter. For **SetGammaRamp**, the first parameter is either D3DSGR_CALIBRATE or D3DSGR_NO_CALIBRATION. The second parameter, *pRamp*, takes a **D3DGAMMARAMP** type. The **D3DGAMMARAMP** type contains three 256-element arrays of **DWORD**s, one

array each to contain the red, green, and blue gamma ramps. **GetGammaRamp** has one parameter that takes a **D3DGAMMARAMP** type that will be filled with the current gamma ramp.
You can include the D3DSGR_CALIBRATE value for the first parameter of **SetGammaRamp** to invoke the calibrator when setting new gamma levels. Calibrating gamma ramps incurs some processing overhead, and should not be used frequently. Setting a calibrated gamma ramp provides a consistent and absolute gamma value for the user, regardless of the display adapter and monitor.
Not all systems support gamma calibration. To determine if gamma calibration is supported, call **Direct3DDevice8.GetDeviceCaps**, and examine the **Caps2** member of the associated **D3DCAPS8** structure after the method returns. If the D3DCAPS2_CANCALIBRATEGAMMA capability flag is present, then gamma calibration is supported.
When setting new ramp levels, keep in mind that that the levels you set in the arrays are only used when your application is in full-screen, exclusive mode. Whenever your application changes to normal mode, the ramp levels are set aside, taking effect again when the application reinstates full-screen mode.

Accessing Surface Memory Directly

[C++]
You can directly access the surface memory by using the **IDirect3DSurface8::LockRect** method. When you call this method, the *pRect* parameter is a pointer to a **RECT** structure that describes the rectangle on the surface to access directly. To request that the entire surface be locked, set *pRect* to NULL. Also, you can specify a **RECT** that covers only a portion of the surface. Providing that no two rectangles overlap, two threads or processes can simultaneously lock multiple rectangles in a surface. Note that a multisample back buffer cannot be locked.
The **LockRect** method fills a **D3DLOCKED_RECT** structure with all the information to properly access the surface memory. The structure includes information about the pitch and has a pointer to the locked bits. When you finish accessing the surface memory, call the **IDirect3DSurface8::UnlockRect** method to unlock it.
While you have a surface locked, you can directly manipulate the contents. The following list describes some tips for avoiding common problems with directly rendering surface memory.

- Never assume a constant display pitch. Always examine the pitch information returned by the **LockRect** method. This pitch can vary for a number of reasons, including the location of the surface memory, the display card type, or even the version of the Microsoft® Direct3D® driver. For more information, see Width vs. Pitch.
- Make certain you copy to unlocked surfaces. Direct3D copy methods will fail if called on a locked surface.
- Limit your application's activity while a surface is locked.

- Always copy data aligned to display memory. Microsoft® Windows® 95 and Windows 98 use a page fault handler, Vflatd.386, to implement a virtual flat-frame buffer for display cards with bank-switched memory. The handler allows these display devices to present a linear frame buffer to Direct3D. Copying data unaligned to display memory can cause the system to suspend operations if the copy spans memory banks.
- A surface may not be locked if it belongs to a resource assigned to the D3DPOOL_DEFAULT memory pool. Back buffer surfaces, which may be accessed using the **IDirect3DDevice8::GetBackBuffer** and **IDirect3DSwapChain8::GetBackBuffer** methods, may be locked only if the swap chain was created with the **Flags** member of the **D3DPRESENT_PARAMETERS** structure set to include D3DPRESENTFLAG_LOCKABLE_BACKBUFFER.

[Visual Basic]
You can directly access the surface memory by using the **Direct3DSurface8.LockRect** method. When you call this method, the *RECT* parameter is a **RECT** type that describes the rectangle on the surface to access directly. To request that the entire surface be locked, set *RECT* to Nothing. Also, you can specify a **RECT** that covers only a portion of the surface. Providing that no two rectangles overlap, two threads or processes can simultaneously lock multiple rectangles in a surface. Note that a multisample back buffer cannot be locked.
The **LockRect** method fills a **D3DLOCKED_RECT** type with all the information to properly access the surface memory. The structure includes information about the pitch and has the locked bits. When you finish accessing the surface memory, call the **Direct3DSurface8.UnlockRect** method to unlock it.
While you have a surface locked, you can directly manipulate the contents. The following list describes some tips for avoiding common problems with directly rendering surface memory.

- Never assume a constant display pitch. Always examine the pitch information returned by the **LockRect** method. This pitch can vary for a number of reasons, including the location of the surface memory, the display card type, or even the version of the Microsoft® Direct3D® driver. For more information, see Width vs. Pitch.
- Make certain you copy to unlocked surfaces. Direct3D copy methods will fail if called on a locked surface.
- Limit your application's activity while a surface is locked.
- Always copy data aligned to display memory. Microsoft® Windows® 95 and Windows 98 use a page fault handler, Vflatd.386, to implement a virtual flat-frame buffer for display cards with bank-switched memory. The handler allows these display devices to present a linear frame buffer to Direct3D. Copying data unaligned to display memory can cause the system to suspend operations if the copy spans memory banks.

- A surface may not be locked if it belongs to a resource assigned to the D3DPOOL_DEFAULT memory pool. Back buffer surfaces, which may be accessed using the **Direct3DDevice8.GetBackBuffer** and **Direct3DSwapChain8.GetBackBuffer** methods, may be locked only if the swap chain was created with the **Flags** member of the **D3DPRESENT_PARAMETERS** structure set to include D3DPRESENTFLAG_LOCKABLE_BACKBUFFER.

Private Surface Data

[C++]
You can store any kind of application-specific data with a surface. For example, a surface representing a map in a game might contain information about terrain.
A surface can have more than one private data buffer. Each buffer is identified by a GUID that you supply when attaching the data to the surface.
To store private surface data, use the **IDirect3DSurface8::SetPrivateData** method, passing a pointer to the source buffer, the size of the data, and an application-defined GUID for the data. Optionally, the source data can exist in the form of a COM object; in this case, you pass a pointer to the object's **IUnknown** interface pointer and you set the D3DSPD_IUNKNOWNPOINTER flag.
SetPrivateData allocates an internal buffer for the data and copies it. You can then safely free the source buffer or object. The internal buffer or interface reference is released when **IDirect3DSurface8::FreePrivateData** is called. This happens automatically when the surface is freed.
To retrieve private data for a surface, you must allocate a buffer of the correct size and then call the **IDirect3DSurface8::GetPrivateData** method, passing the GUID that was assigned to the data by **SetPrivateData**. You are responsible for freeing any dynamic memory you use for this buffer. If the data is a COM object, this method retrieves the **IUnknown** pointer.
If you don't know how big a buffer to allocate, first call **GetPrivateData** with zero in *SizeOfData*. If the method fails with D3DERR_MOREDATA, it returns the necessary number of bytes for the buffer in *SizeOfData*.

[Visual Basic]
You can store any kind of application-specific data with a surface. For example, a surface representing a map in a game might contain information about terrain.
A surface can have more than one private data buffer. Each buffer is identified by a GUID that you supply when attaching the data to the surface.
To store private surface data, use the **Direct3DSurface8.SetPrivateData** method, passing a buffer that contains data to associate with the surface, the size of the data buffer, and an application-defined DXGUID for the data.
SetPrivateData allocates an internal buffer for the data and copies it. You can then safely free the source buffer or object. The internal buffer or interface reference is

released when **Direct3DSurface8.FreePrivateData** is called. This happens automatically when the surface is freed.
To retrieve private data for a surface, you must allocate a buffer of the correct size and then call the **Direct3DSurface8.GetPrivateData** method, passing the DXGUID that was assigned to the data by **SetPrivateData**. You are responsible for freeing any dynamic memory you use for this buffer.
If you don't know how big a buffer to allocate, first call **GetPrivateData** with zero in *SizeOfData.* If the method fails with D3DERR_MOREDATA, it returns the necessary number of bytes for the buffer in *SizeOfData*.

Lights

Lights are used to illuminate objects in a scene. This section describes lights and how they are used in Microsoft® Direct3D® applications. The following topics are discussed.

- Introduction to Lighting and Materials
- Direct3D Light Model vs. Nature
- Color Values for Lights and Materials
- Direct Light vs. Ambient Light
- Light Objects
- Light Properties
- Using Lights

Introduction to Lighting and Materials

When lighting is enabled, as Microsoft® Direct3D® rasterizes a scene in the final stage of rendering, it determines the color of each rendered pixel based on a combination of the current material color and the texels in an associated texture map; the diffuse and specular colors at the vertex, if specified; and the color and intensity of light produced by light sources in the scene or the scene's ambient light level.
When you use Direct3D lighting and materials, you allow Direct3D to handle the details of illumination for you. Advanced users can perform lighting on their own, if desired.
How you work with lighting and materials makes a big difference in the appearance of the rendered scene. Materials define how light reflects off a surface. Direct light and ambient light levels define the light that is reflected. You must use materials to render a scene if lighting is enabled. Lights are not required to render a scene, but details in a scene rendered without light are not visible. At best, rendering an unlit scene results in a silhouette of the objects in the scene. This is not enough detail for most purposes.

Direct3D Light Model vs. Nature

In nature, when light is emitted from a source, it is reflect off of hundreds, if not thousands or millions, of objects before reaching the user's eye. Each time it is reflected, some light is absorbed by a surface, some is scattered in random directions, and the rest goes on to another surface or to the user's eye. This process continues until the light is reduced to nothing or a user perceives the light.
Obviously, the calculations required to perfectly simulate the natural behavior of light are too time-consuming to use for real-time 3-D graphics. Therefore, with speed in mind, the Microsoft® Direct3D® light model approximates the way light works in the natural world. Direct3D describes light in terms of red, green, and blue components that combine to create a final color. For more information, see Color Values for Lights and Materials.
In Direct3D, when light reflects off a surface, the light color interacts mathematically with the surface itself to create the color eventually displayed on the screen. For specific information about the algorithms Direct3D uses, see Mathematics of Direct3D Lighting.
The Direct3D light model generalizes light into two types: ambient light and direct light. Each has different attributes, and each interacts with the material of a surface in different ways. Ambient light is light that has been scattered so much that its direction and source are indeterminate: it maintains a low-level of intensity everywhere. The indirect lighting used by photographers is a good example of ambient light. Ambient light in Direct3D, as in nature, has no real direction or source, only a color and intensity. In fact, the ambient light level is completely independent of any objects in a scene that generate light. Ambient light does not contribute to specular reflection.
Direct light is the light generated by a source within a scene; it always has color and intensity, and it travels in a specified direction. Direct light interacts with the material of a surface to create specular highlights, and its direction is used as a factor in shading algorithms, including Gouraud shading. When direct light is reflected, it does not contribute to the ambient light level in a scene. The sources in a scene that generate direct light have different characteristics that affect how they illuminate a scene. For more information, see Lights.
Additionally, a polygon's material has properties that affect how that polygon reflects the light it receives. You set a single reflectance trait that describes how the material reflects ambient light, and you set individual traits to determine the material's specular and diffuse reflectance. For more information, see Materials.

Color Values for Lights and Materials

[C++]
Microsoft® Direct3D® describes color in terms of four components—red, green, blue, and alpha—that combine to make a final color. The **D3DCOLORVALUE** C++ structure is defined to contain values for each component. Each member is a floating-point value that typically ranges from 0.0 to 1.0, inclusive. Although both lights and materials use the same structure to describe color, the values in the structure are used a little differently by each.

[Visual Basic]

Microsoft® Direct3D® describes color in terms of four components—red, green, blue, and alpha—that combine to make a final color. The **D3DCOLORVALUE** type is defined to contain values for each component. Each member is a floating-point value that typically ranges from 0.0 to 1.0, inclusive. Although both lights and materials use the same structure to describe color, the values in the structure are used a little differently by each.

Color values for light sources represent the amount of a particular light component it emits. Because lights don't use an alpha component, only the red, green, and blue components of the color are relevant. You can visualize the three components as the red, green, and blue lenses on a projection television. Each lens might be off (a 0.0 value in the appropriate member), it might be as bright as possible (a 1.0 value), or it might be some level in between. The colors coming through the lenses combine to make the light's final color. A combination like R: 1.0, G: 1.0, B: 1.0 creates a white light, where R: 0.0, G: 0.0, B: 0.0 doesn't emit light at all. You can make a light that emits only one component, resulting in a pure red, green, or blue light, or, the light could use combinations to emit colors like yellow or purple. You can even set negative color component values to create a "dark light" that actually removes light from a scene. Or, you might set the components to some value larger than 1.0 to create an extremely bright light.
With materials, on the other hand, color values represent how much of a light component is reflected by a surface that is rendered with that material. A material whose color components are R: 1.0, G: 1.0, B: 1.0, A: 1.0 reflects all the light that comes its way. Likewise, a material with R: 0.0, G: 1.0, B: 0.0, A: 1.0 reflects all the green light that is directed at it. Materials have multiple reflectance values to create various types of effects; for more information, see Material Properties.
Color values for ambient light are different from those used for direct light sources and materials. For more information, see Direct Light vs. Ambient Light.

Direct Light vs. Ambient Light

Although both direct and ambient light illuminate objects in a scene, they are independent of one another, they have very different effects, and they require that you work with them in completely different ways.

[C++]
Direct light is just that: direct. Direct light always has direction and color, and it is a factor for shading algorithms, such as Gouraud shading. Different types of lights emit direct light in different ways, creating special attenuation effects. You create a set of light parameters for direct light by calling the **IDirect3DDevice8::SetLight** method. For more information, see Lights.
Ambient light is effectively everywhere in a scene. You can think of it as a general level of light that fills an entire scene, regardless of the objects and their locations in that scene. Ambient light, being everywhere, has no position or direction, only color and intensity. Set the ambient light level with a call to the **IDirect3DDevice8::SetRenderState** method, specifying D3DRS_AMBIENT as the *State* parameter, and the desired RGBA color as the *Value* parameter.

Ambient light color takes the form of an RGBA value, where each component is an integer value from 0 to 255. This is unlike most color values in Microsoft® Direct3D®. For more information, see Color Values for Lights and Materials.
You can use the **D3DCOLOR_RGBA** macro to generate RGBA values. The red, green, and blue components combine to make the final color of the ambient light. The alpha component controls the transparency of the color. When using hardware acceleration or RGB emulation, the alpha component is ignored.

[Visual Basic]
Direct light is just that: direct. Direct light always has direction and color, and it is a factor for shading algorithms, such as Gouraud shading. Different types of lights emit direct light in different ways, creating special attenuation effects. You create a set of light parameters for direct light by calling the **Direct3DDevice8.SetLight** method. For more information, see Lights.
Ambient light is effectively everywhere in a scene. You can think of it as a general level of light that fills an entire scene, regardless of the objects and their locations in that scene. Ambient light, being everywhere, has no position or direction, only color and intensity. Set the ambient light level with a call to the **Direct3DDevice8.SetRenderState** method, specifying D3DRS_AMBIENT as the *State* parameter, and the desired RGBA color as the *Value* parameter.

Light Objects

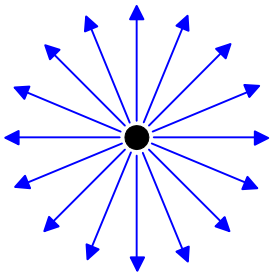
Microsoft® Direct3D® employs three types of lights: point lights, spotlights, and directional lights. You choose the type of light you want when you create a set of light properties. The illumination properties and the resulting computational overhead varies with each type of light source. The following types of light sources, supported by the Direct3D lighting module, are discussed.

- Point Lights
- Spotlights
- Directional Lights

Do not confuse light sources in a scene with the concept of an ambient light level. For more information, see Direct Light vs. Ambient Light, Light Properties and Using Lights.

Point Lights

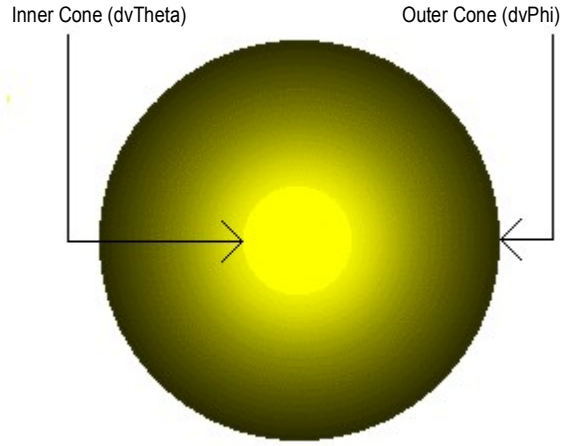
Point lights have color and position within a scene, but no single direction. They give off light equally in all directions, as shown in the following illustration.



A light bulb is a good example of a point light. Point lights are affected by attenuation and range, and illuminate a mesh on a vertex-by-vertex basis. During lighting, Microsoft® Direct3D® uses the point light's position in world space and the coordinates of the vertex being lit to derive a vector for the direction of the light, and the distance that the light has traveled. Both are used, along with the vertex normal, to calculate the contribution of the light to the illumination of the surface.

Spotlights

Spotlights have color, position, and direction in which they emit light. Light emitted from a spotlight is made up of a bright inner cone and a larger outer cone, with the light intensity diminishing between the two, as shown in the following illustration.



Spotlights are affected by falloff, attenuation, and range. These factors, as well as the distance light travels to each vertex, are figured in when computing lighting effects for objects in a scene. Computing these effects for each vertex makes spotlights the most computationally expensive of all lights in Microsoft Direct3D®.

Directional Lights

Directional lights have only color and direction, not position. They emit parallel light. This means that all light generated by directional lights travels through a scene in the same direction. Imagine a directional light as a light source at near infinite distance, such as the sun. Directional lights are not affected by attenuation or range, so the direction and color you specify are the only factors considered when Microsoft® Direct3D® calculates vertex colors. Because of the small number of illumination factors, these are the least computationally intensive lights to use.

Light Properties

[C++]
Light properties describe a light source's type and color. Depending on the type of light being used, a light can have properties for attenuation and range, or for spotlight effects. But, not all types of lights use all properties. Microsoft® Direct3D® uses the **D3DLIGHT8** structure to carry information about light properties for all types of light sources. This section contains information for all light properties. Information is divided into the following groups.

[Visual Basic]
Light properties describe a light source's type and color. Depending on the type of light being used, a light can have properties for attenuation and range, or for spotlight effects. But, not all types of lights use all properties. Microsoft® Direct3D® uses the **D3DLIGHT8** type for Microsoft Visual Basic® to carry information about light properties for all types of light sources. This section contains information for all light properties. Information is divided into the following groups.

- Light Type
- Light Color
- Color Vertices
- Light Position, Range, and Attenuation
- Light Direction
- Spotlight Properties

Light properties affect how a light source illuminates objects in a scene.
For more information, see Using Lights, Setting Light Properties, and Mathematics of Direct3D Lighting.

Light Type

[C++]
The light type property defines which type of light source you're using. The light type is set by using a value from the **D3DLIGHTTYPE** C++ enumeration in the **Type**

member of the light's **D3DLIGHT8** structure. There are three types of lights in Microsoft® Direct3D®—point lights, spotlights, and directional lights. Each type illuminates objects in a scene differently, with varying levels of computational overhead. For general information about how each type of light works, see Light Objects.

[Visual Basic]
The light type property defines which type of light source you're using. The light type is set by using a value from the **CONST_D3DLIGHTTYPE** Microsoft Visual Basic® enumeration in the **Type** member of the light's **D3DLIGHT8** type. There are three types of lights in Microsoft® Direct3D®—point lights, spotlights, and directional lights. Each type illuminates objects in a scene differently, with varying levels of computational overhead. For general information about how each type of light works, see Light Objects.

Light Color

Lights in Microsoft® Direct3D® emit three colors that are used independently in the system's lighting computations: a diffuse color, an ambient color, and a specular color. Each is incorporated by the Direct3D lighting module, interacting with a counterpart from the current material, to produce a final color used in rendering. The diffuse color interacts with the diffuse reflectance property of the current material, the specular color with the material's specular reflectance property, and so on. For specifics about how Direct3D applies these colors, see Mathematics of Direct3D Lighting.

[C++]
In a C++ application, the **D3DLIGHT8** structure includes three members for these colors—**Diffuse**, **Ambient**, and **Specular**—each one is a **D3DCOLORVALUE** structure that defines the color being emitted.

[Visual Basic]
The **D3DLIGHT8** Microsoft Visual Basic® type includes three members for these colors—**Diffuse**, **Ambient**, and **Specular**—each one is a **D3DCOLORVALUE** type that defines the color being emitted.

The type of color that applies most heavily to the system's computations is the diffuse color. The most common diffuse color is white (R:1.0 G:1.0 B:1.0), but you can create colors as needed to achieve desired effects. For example, you could use red light for a fireplace, or you could use green light for a traffic signal set to "Go." Generally, you set the light color components to values between 0.0 and 1.0, inclusive, but this isn't a requirement. For example, you might set all the components to 2.0, creating a light that is "brighter than white." This type of setting can be especially useful when you use attenuation settings other than constant.

Note that although Direct3D uses RGBA values for lights, the alpha color component is not used. For more information, see Color Values for Lights and Materials.

Color Vertices

[C++]
Usually material colors are used for lighting. However, you can specify that material colors—emissive, ambient, diffuse, and specular—are to be overridden by diffuse or specular vertex colors. This is done by calling **IDirect3DDevice8::SetRenderState** and setting the device state variables listed in the following table.

[Visual Basic]
Usually material colors are used for lighting. However, you can specify that material colors—emissive, ambient, diffuse, and specular—are to be overridden by diffuse or specular vertex colors. This is done by calling **Direct3DDevice8.SetRenderState** and setting the device state variables listed in the following table.

Device state variable	Meaning	Type
D3DRS_AMBIENTMATERIALSOURCE	Defines where to get ambient material color.	D3DMATERIAL
D3DRS_DIFFUSEMATERIALSOURCE	Defines where to get diffuse material color.	D3DMATERIAL
D3DRS_SPECULARMATERIALSOURCE	Defines where to get specular material color.	D3DMATERIAL
D3DRS_EMISSIVEMATERIALSOURCE	Defines where to get emissive material color.	D3DMATERIAL
D3DRS_COLORVERTEX	Disables or enables use of vertex colors.	BOOL

The alpha/transparency value always comes only from the diffuse color's alpha channel.
The fog value always comes only from the specular color's alpha channel.
D3DMATERIALCOLORSOURCE can have the following values.

- D3DMCS_MATERIAL - Material color is used as source.
- D3DMCS_COLOR1 - Diffuse vertex color is used as source.
- D3DMCS_COLOR2 - Specular vertex color is used as source.

Light Position, Range, and Attenuation

[C++]

The position, range, and attenuation properties define a light's location in world space, and how the light it emits behaves over distance. As with all light properties you use in C++, these are contained in a light's **D3DLIGHT8** structure.

Position

Light position is described using a **D3DVECTOR** structure in the **Position** member of the **D3DLIGHT8** structure. The x-, y-, and z-coordinates are assumed to be in world space. Directional lights are the only type of light that don't use the position property.

Range

A light's range property determines the distance, in world space, at which meshes in a scene no longer receive light emitted by that object. The **Range** member contains a floating-point value that represents the light's maximum range, in world space. Directional lights don't use the range property.

Attenuation

Attenuation controls how a light's intensity decreases toward the maximum distance specified by the range property. Three **D3DLIGHT8** structure members represent light attenuation: **Attenuation0**, **Attenuation1**, and **Attenuation2**. These members contain floating-point values ranging from 0.0 through infinity, controlling a light's attenuation. Some applications set the **Attenuation1** member to 1.0 and the others to 0.0, resulting in light intensity that changes as 1 / *D*, where *D* is the distance from the light source to the vertex. The maximum light intensity is at the source, decreasing to 1 / (*Light Range*) at the light's range. Typically, an application sets **Attenuation0** to 0.0, **Attenuation1** to a constant value, and **Attenuation2** to 0.0.

You can combine attenuation values to get more complex attenuation effects. Or, you might set them to values outside the normal range to create even stranger attenuation effects; negative attenuation values make a light that gets brighter over distance.

For more information about the mathematical model that Microsoft® Direct3D® uses to calculate attenuation, see Light Attenuation Over Distance. Like the range property, directional lights don't use the attenuation property.

[Visual Basic]

The **D3DLIGHT8** type includes members that your Microsoft® Visual Basic® application uses to define a light's position, range, and attenuation properties. These describe a light's location in world space, and how the light it emits behaves over distance.

Position

Light position is described using a **D3DVECTOR** type in the **Position** member of the **D3DLIGHT8** type. The x-, y-, and z-coordinates are assumed to be in world space. Directional lights are the only type of light that don't use the position property.

Range

A light's range property determines the distance, in world space, at which meshes in a scene no longer receive light emitted by that object. The **Range** member contains a floating-point value that represents the light's maximum range, in world space. Directional lights don't use the range property.

Attenuation

Attenuation controls how a light's intensity decreases toward the maximum distance specified by the range property. Three **D3DLIGHT8** members represent light

attenuation: **Attenuation0**, **Attenuation1**, and **Attenuation2**. These members contain floating-point values ranging from 0.0 through infinity, controlling a light's attenuation. Some applications set the **Attenuation1** member to 1.0 and the others to 0.0, resulting in light intensity that changes as 1 / *D*, where *D* is the distance from the light source to the vertex. The maximum light intensity is at the source, decreasing to 1 / (*Light Range*) at the light's range. Typically, an application sets **Attenuation0** to 0.0, **Attenuation1** to a constant value, and **Attenuation2** to 0.0.
You can combine attenuation values to get more complex attenuation effects. Or, you might set them to values outside the normal range to create even stranger attenuation effects; negative attenuation values make a light that gets brighter over distance.
For more information about the mathematical model that Microsoft® Direct3D® uses to calculate attenuation, see Light Attenuation Over Distance. Like the range property, directional lights don't use the attenuation property.

Light Direction

A light's direction property determines the direction that the light emitted by the object travels, in world space. Direction is used only by directional lights and spotlights, and is described with a vector.

[C++]
C++ applications set the light direction in the **Direction** member of the light's **D3DLIGHT8** structure. The **Direction** member is of type **D3DVECTOR**. Direction vectors are described as distances from a logical origin, regardless of the light's position in a scene. Therefore, a spotlight that points straight into a scene—along the positive z-axis—has a direction vector of <0,0,1> no matter where its position is defined to be. Similarly, you can simulate sunlight shining directly on a scene by using a directional light whose direction is <0,-1,0>. Obviously, you don't have to create lights that shine along the coordinate axes; you can mix and match values to create lights that shine at more interesting angles.

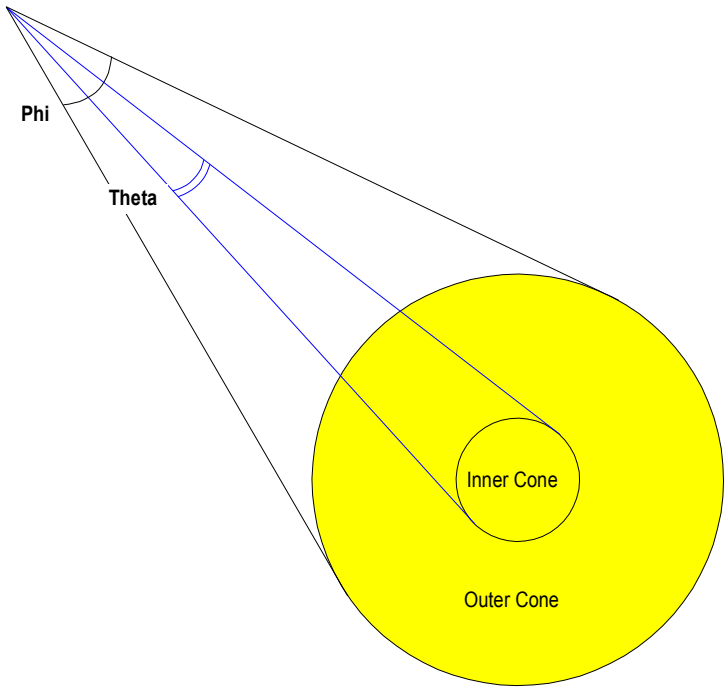
[Visual Basic]
Microsoft® Visual Basic® applications set the light direction in the **Direction** member of the light's **D3DLIGHT8** type. The **Direction** member is of type **D3DVECTOR**. Direction vectors are described as distances from a logical origin, regardless of the light's position within a scene. Therefore, a spotlight that points straight into a scene —along the positive z-axis—has a direction vector of <0,0,1> no matter where its position is defined to be. Similarly, you can simulate sunlight shining directly on a scene by using a directional light whose direction is <0,-1,0>. Obviously, you don't have to create lights that shine along the coordinate axes; you can mix and match values to create lights that shine at more interesting angles.

Note

Although you don't need to normalize a light's direction vector, always be sure that it has magnitude. In other words, don't use a <0,0,0> direction vector.

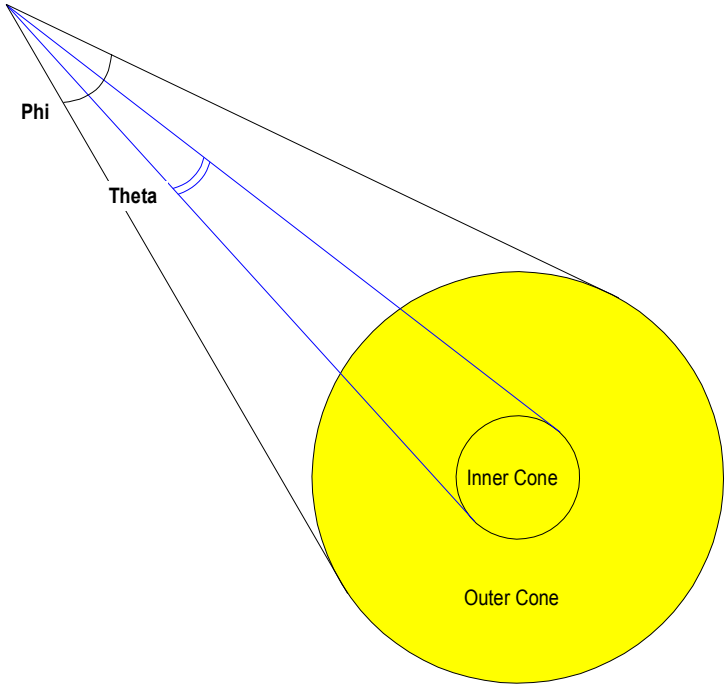
Spotlight Properties

[C++]
The **D3DLIGHT8** C++ structure contains three members that are used only by spotlights. These members—**Falloff**, **Theta**, and **Phi**—control how large or small a spotlight object's inner and outer cones are, and how light decreases between them. For general information about these characteristics, see Spotlights.
The **Theta** value is the radian angle of the spotlight's inner cone, and the **Phi** value is the angle for the outer cone of light. The **Falloff** value controls how light intensity decreases between the outer edge of the inner cone and the inner edge of the outer cone. Most applications set **Falloff** to 1.0 to create falloff that occurs evenly between the two cones, but you can set other values as needed.
For more information about the mathematical model used by Microsoft® Direct3D® for calculating falloff, see Spotlight Falloff Model.
The following illustration shows the relationship between the values for these members and how they can affect a spotlight's inner and outer cones of light.



[Visual Basic]
The **D3DLIGHT8** Microsoft® Visual Basic® type contains three members that are used only by spotlights. These members—**Falloff**, **Theta**, and **Phi**—control how

large or small a spotlight object's inner and outer cones are, and how light decreases between them. For general information about these characteristics, see Spotlights.
The **Theta** value is the radian angle of the spotlight's inner cone, and the **Phi** value is the angle for the outer cone of light. The **Falloff** value controls how light intensity decreases between the outer edge of the inner cone and the inner edge of the outer cone. Most applications set **Falloff** to 1.0 to create falloff that occurs evenly between the two cones, but you can set other values as needed.
For more information about the mathematical model used by Microsoft® Direct3D® for calculating falloff, see Spotlight Falloff Model.
The following illustration shows the relationship between the values for these members and how they can affect a spotlight's inner and outer cones of light.



Using Lights

This section provides information about using lights in a Microsoft® Direct3D® application. Information is divided into the following topics.

- Setting Light Properties
- Enabling and Disabling Lights
- Retrieving Light Properties

Setting Light Properties

[C++]
You set lighting properties in a C++ application by preparing a **D3DLIGHT8** structure and then calling the **IDirect3DDevice8::SetLight** method. The **SetLight** method accepts the index at which the device should place the set of light properties to its internal list of light properties, and the address of a prepared **D3DLIGHT8** structure that defines those properties. You can call **SetLight** with new information as needed to update the light's illumination properties.
The system allocates memory to accommodate a set of lighting properties each time you call the **SetLight** method with an index that has never been assigned properties. Applications can set a number of lights, with only a subset of the assigned lights enabled at a time. Check the **MaxActiveLights** member of the **D3DCAPS8** structure when you retrieve device capabilities to determine the maximum number of active lights supported by that device. If you no longer need a light, you can disable it or overwrite it with a new set of light properties.
The following C++ code example prepares and sets properties for a white point-light whose emitted light will not attenuate over distance.

```
/*
 * For the purposes of this example, the d3dDevice variable
 * is a valid pointer to an IDirect3DDevice8 interface.
 */
D3DLIGHT8 d3dLight;
HRESULT   hr;

// Initialize the structure.
ZeroMemory(&d3dLight, sizeof(D3DLIGHT8));

// Set up a white point light.
d3dLight.Type = D3DLIGHT_POINT;
d3dLight.Diffuse.r  = 1.0f;
d3dLight.Diffuse.g  = 1.0f;
d3dLight.Diffuse.b  = 1.0f;
d3dLight.Ambient.r  = 1.0f;
d3dLight.Ambient.g  = 1.0f;
d3dLight.Ambient.b  = 1.0f;
d3dLight.Specular.r = 1.0f;
d3dLight.Specular.g = 1.0f;
d3dLight.Specular.b = 1.0f;

// Position it high in the scene and behind the user.
// Remember, these coordinates are in world space, so
// the user could be anywhere in world space, too.
// For the purposes of this example, assume the user
// is at the origin of world space.
d3dLight.Position.x = 0.0f;
```

```
d3dLight.Position.y = 1000.0f;
d3dLight.Position.z = -100.0f;

// Don't attenuate.
d3dLight.Attenuation0 = 1.0f;
d3dLight.dvRange      = 1000.0f;

// Set the property information for the first light.
hr = d3dDevice->SetLight(0, &d3dLight);
if (FAILED(hr))
{
    // Code to handle the error goes here.
}
```

You can update a set of light properties with another call to **SetLight** at any time. Just specify the index of the set of light properties to update and the address of the **D3DLIGHT8** structure that contains the new properties.

Note

Assigning a set of light properties to the device does not enable the light source whose properties are being added. Enable a light source by calling the **IDirect3DDevice8::LightEnable** method for the device.

[Visual Basic]

Your Microsoft Visual Basic® application sets lighting properties by preparing a **D3DLIGHT8** type and then calling the **Direct3DDevice8.SetLight** method. The **SetLight** method accepts the index at which the device should place the set of light properties to its internal list of light properties, and the address of a prepared **D3DLIGHT8** type that defines those properties. You can call **SetLight** with new information as needed to update the light's illumination properties.

The system allocates memory to accommodate a set of lighting properties each time you call the **SetLight** method with an index that has never had properties assigned. Applications can set a number of lights, with only a subset of the assigned lights enabled at a time. Check the **MaxActiveLights** member of the **D3DCAPS8** type when you retrieve device capabilities to determine the maximum number of active lights supported by that device. If you no longer need a light, you can disable it or overwrite it with a new set of light properties.

The following code example prepares and sets properties for a white point-light whose emitted light will not attenuate over distance.

```
'
' For this example, the d3dDevice variable contains a valid reference
' to a Direct3DDevice8 object.
'
Dim LightDesc As D3DLIGHT8
Dim c As D3DCOLORVALUE
Dim vPos As D3DVECTOR
```

```
' Use the same color settings for all emitted light color.
With c
   .r = 1#: .g = 1#: .b = 1#
   .a = 1# ' The alpha component isn't used for lights.
End With

With vPos
   .x = 0: .y = 1000: .z = -100
End With

With LightDesc
   .Type = D3DLIGHT_POINT
   .Position = vPos
   .Ambient = c: .diffuse = c: .specular = c
   .Attenuation0 = 1# ' Don't attenuate the light
End With

d3dDevice.SetLight 0, LightDesc
```

You can update a set of light properties with another call to **SetLight** at any time. Specify the index of the set of light properties to update and the **D3DLIGHT8** type that contains the new properties.

Note

Assigning a set of light properties to the device does not enable the light source whose properties are being added. Enable a light source by calling the **Direct3DDevice8.LightEnable** method for the device.

Enabling and Disabling Lights

[C++]

Once you assign a set of light properties for a light source in a scene, the light source can be activated by calling the **IDirect3DDevice8::LightEnable** method for the device. New light sources are disabled by default. The **LightEnable** method accepts two parameters. Set the first parameter to the zero-based index of the light source to be affected by the method, and set the second parameter to TRUE to enable the light or FALSE to disable it.

The following code example illustrates the use of this method by enabling the first light source in the device's list of light source properties.

```
/*
 * For the purposes of this example, the d3dDevice variable
 * is a valid pointer to an IDirect3DDevice8 interface.
 */
HRESULT hr;
```

```
hr = pd3dDevice->LightEnable(0, TRUE);
if (FAILED(hr))
{
    // Code to handle the error goes here.
}
```

[Visual Basic]

Once you assign a set of light properties for a light source in a scene, the light source can be activated by calling the **Direct3DDevice8.LightEnable** method for the device. New light sources are disabled by default. The **LightEnable** method accepts two parameters. Set the first parameter to the zero-based index of the light source to be affected by the method, and set the second parameter to True to enable the light or False to disable it.

The following code example illustrates the use of this method by enabling the first light source in the device's list of light source properties.

```
'
' For the purposes of this example, the d3dDevice variable contains
' a valid reference to Direct3DDevice8 object.
'
On Local Error Resume Next

Call d3dDevice.LightEnable(0, True)
If Err.Number <> D3D_OK Then
    'Code to handle the error goes here.
End If
```

[C++]

If you enable or disable a light that has no properties that are set with **IDirect3DDevice8::SetLight**, the **LightEnable** method creates a light source with the properties listed in following table and enables or disables it.

[Visual Basic]

If you enable or disable a light that has no properties that are set with **Direct3Ddevice8.SetLight**, the **LightEnable** method creates a light source with the properties listed in following table and enables or disables it.

Member	**Default**
Type	D3DLIGHT_DIRECTIONAL
Diffuse	(R:1, G:1, B:1, A:0)

Specular	(R:0, G:0, B:0, A:0)
Ambient	(R:0, G:0, B:0, A:0)
Position	(0, 0, 0)
Direction	(0, 0, 1)
Range	0
Falloff	0
Attenuation0	0
Attenuation1	0
Attenuation2	0
Theta	0
Phi	0

Retrieving Light Properties

[C++]
You can retrieve all the properties for an existing light source from C++ by calling the **IDirect3DDevice8::GetLight** method for the device. When calling the **GetLight** method, pass in the first parameter the zero-based index of the light source for which the properties will be retrieved, and supply the address of a **D3DLIGHT8** structure in the second parameter. The device fills the **D3DLIGHT8** structure to describe the lighting properties it uses for the light source at that index.
The following code example illustrates this process.

```
/*
 * For the purposes of this example, the pd3dDevice variable
 * is a valid pointer to an IDirect3DDevice8 interface.
 */
HRESULT hr;
D3DLIGHT8 light;

// Get the property information for the first light.
hr = pd3dDevice->GetLight(0, &light);
if (FAILED(hr))
{
    // Code to handle the error goes here.
}
```

If you supply an index outside the range of the light sources assigned in the device, the **GetLight** method fails, returning D3DERR_INVALIDCALL.

[Visual Basic]
A Microsoft® Visual Basic® application retrieves the properties for an existing light source by calling the **Direct3DDevice8.GetLight** method for the device. When calling the **GetLight** method, pass in the first parameter the zero-based index of the

light source for which the properties will be retrieved, and a variable of type **D3DLIGHT8** as the second parameter. The device fills the **D3DLIGHT8** type to describe the lighting properties it uses for the light source at that index.
The following code example illustrates this process.

```
'
' For the purposes of this example, the d3dDevice variable contains
' a valid reference to a Direct3DDevice8 object.
'
Dim lightDesc As D3DLIGHT8

' Get the property information for the first light.
Call d3dDevice.GetLight(0, lightDesc)
If Err.Number <> D3D_OK Then
    ' Code to handle the error goes here.
End If
```

If you supply an index outside the range of the light sources assigned in the device, the **GetLight** method fails, and the value of **Err.Number** is D3DERR_INVALIDCALL.

Materials

This section describes materials and how they are used in Microsoft® Direct3D® applications. The following topics are discussed.

- What Are Materials?
- Material Properties
- Using Materials

Default Material Properties Note

If your application does not specify material properties for rendering, the system uses a default material. The default material reflects all diffuse light—white, for example—with no ambient or specular reflection, and no emissive color.

What Are Materials?

Materials describe how polygons reflect light or appear to emit light in a 3-D scene. Essentially, a material is a set of properties that tell Microsoft® Direct3D® the following things about the polygons it is rendering.

- How they reflect ambient and diffuse light
- What their specular highlights look like
- Whether the polygons appear to emit light

[C++]

Direct3D applications written in C++ use the **D3DMATERIAL8** structure to describe material properties. For more information, see Material Properties.

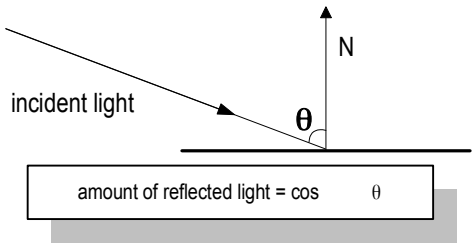
[Visual Basic]
Direct3D applications written in Microsoft® Visual Basic® use the **D3DMATERIAL8** type to describe material properties. For more information, see Material Properties.

Material Properties

[C++]
Material properties detail a material's diffuse reflection, ambient reflection, light emission, and specular highlight characteristics. Microsoft® Direct3D® uses the **D3DMATERIAL8** structure to carry all material property information. Material properties affect the colors that Direct3D uses to rasterize polygons that use the material. With the exception of the specular property, each property is described as an RGBA color that represents how much of the red, green, and blue parts of a given type of light it reflects, and an alpha blending factor—the alpha component of the RGBA color. The material's specular property is described in two parts: color and power. For more information, see Color Values for Lights and Materials.

Diffuse and Ambient Reflection

The **Diffuse** and **Ambient** members of the **D3DMATERIAL8** structure describe how a material reflects the ambient and diffuse light in a scene. Because most scenes contain much more diffuse light than ambient light, diffuse reflection plays the largest part in determining color. Additionally, because diffuse light is directional, the angle of incidence for diffuse light affects the overall intensity of the reflection. Diffuse reflection is greatest when the light strikes a vertex parallel to the vertex normal. As the angle increases, the effect of diffuse reflection diminishes. The amount of light reflected is the cosine of the angle between the incoming light and the vertex normal, as shown here.



Ambient reflection, like ambient light, is nondirectional. Ambient reflection has a lesser impact on the apparent color of a rendered object, but it does affect the overall color and is most noticeable when little or no diffuse light reflects off the material. A

material's ambient reflection is affected by the ambient light set for a scene by calling the **IDirect3DDevice8::SetRenderState** method with the D3DRS_AMBIENT flag. Diffuse and ambient reflection work together to determine the perceived color of an object, and are usually identical values. For example, to render a blue crystalline object, you create a material that reflects only the blue component of diffuse and ambient light. When placed in a room with a white light, the crystal appears to be blue. However, in a room that has only red light, the same crystal would appear to be black, because its material doesn't reflect red light.

Emission

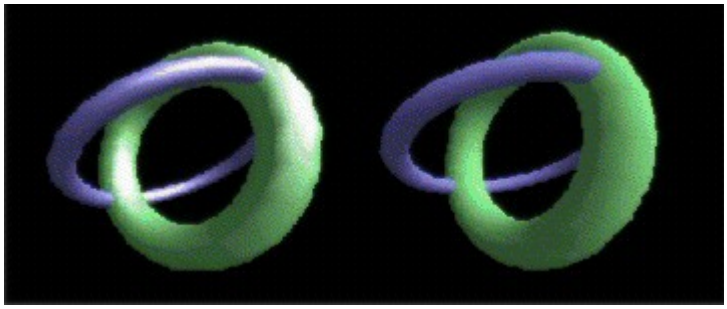
Materials can be used to make a rendered object appear to be self-luminous. The **Emissive** member of the **D3DMATERIAL8** structure is used to describe the color and transparency of the emitted light. Emission affects an object's color and can, for example, make a dark material brighter and take on part of the emitted color.

You can use a material's emissive property to add the illusion that an object is emitting light, without incurring the computational overhead of adding a light to the scene. In the case of the blue crystal, the emissive property is useful if you want to make the crystal appear to light up, but not cast light on other objects in the scene. Remember, materials with emissive properties don't emit light that can be reflected by other objects in a scene. To achieve this reflected light, you need to place an additional light within the scene.

Specular Reflection

Specular reflection creates highlights on objects, making them appear shiny. The **D3DMATERIAL8** structure contains two members that describe the specular highlight color as well as the material's overall shininess. You establish the color of the specular highlights by setting the **Specular** member to the desired RGBA color—the most common colors are white or light gray. The values you set in the **Power** member control how sharp the specular effects are.

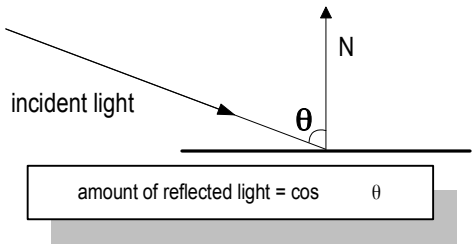
Specular highlights can create dramatic effects. Drawing again on the blue crystal analogy: a larger **Power** value creates sharper specular highlights, making the crystal appear to be quite shiny. Smaller values increase the area of the effect, creating a dull reflection that make the crystal look frosty. To make an object truly matte, set the **Power** member to zero and the color in **Specular** to black. Experiment with different levels of reflection to produce a realistic appearance for your needs. The following illustration shows two identical models. The one on the left uses a specular reflection power of 10; the model on the right has no specular reflection.



[Visual Basic]
Material properties detail a material's diffuse reflection, ambient reflection, light emission, and specular highlight characteristics. Microsoft® Direct3D® uses the **D3DMATERIAL8** type to carry all material property information. Material properties affect the colors Direct3D uses to rasterize polygons that use the material. With the exception of the specular property, each property is described as an RGBA color that represents how much of the red, green, and blue parts of a given type of light it reflects, and an alpha blending factor—the alpha component of the RGBA color. The material's specular property is described in two parts: color and power. For more information, see Color Values for Lights and Materials.

Diffuse and Ambient Reflection

The **diffuse** and **ambient** members of the **D3DMATERIAL8** type describe how a material reflects the ambient and diffuse light in a scene. Because most scenes contain much more diffuse light than ambient light, diffuse reflection plays the largest part in determining color. Additionally, because diffuse light is directional, the angle of incidence for diffuse light affects the overall intensity of the reflection. Diffuse reflection is greatest when the light strikes a vertex parallel to the vertex normal. As the angle increases, the effect of diffuse reflection diminishes. The amount of light reflected is the cosine of the angle between the incoming light and the vertex normal, as shown here.



Ambient reflection, like ambient light, is nondirectional. Ambient reflection has a lesser impact on the apparent color of a rendered object, but it does affect the overall color and is most noticeable when little or no diffuse light reflects off the material. A material's ambient reflection is affected by the ambient light set for a scene by calling the **Direct3DDevice8.SetRenderState** method with the D3DRS_AMBIENT flag. Diffuse and ambient reflection work together to determine the perceived color of an object, and are usually identical values. For example, to render a blue crystalline object, create a material that reflects only the blue component of diffuse and ambient light. When placed in a room with a white light, the crystal appears to be blue. However, in a room that has only red light, the same crystal appears to be black, because its material doesn't reflect red light.

Emission

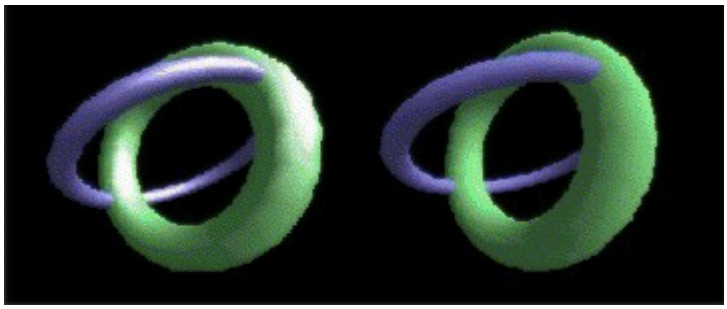
Materials can be used to make a rendered object appear self-luminous. The **emissive** member of the **D3DMATERIAL8** type is used to describe the color and transparency of the emitted light. Emission affects an object's color and can, for example, make a dark material brighter and take on part of the emitted color.

You can use a material's emissive property to add the illusion that an object is emitting light, without incurring the computational overhead of adding a light to the scene. In the case of the blue crystal, the emissive property is useful to make the crystal appear to light up but not cast light on other objects in the scene. Remember, materials with emissive properties don't emit light that can be reflected by other objects in a scene. To achieve this reflected light, you must place an additional light within the scene.

Specular Reflection

Specular reflection creates highlights on objects, making them appear shiny. The **D3DMATERIAL8** type contains two members that describe the specular highlight color as well as the material's overall shininess. You establish the color of the specular highlights by setting the **specular** member to the desired RGBA color—the most common colors are white or light gray. The values you set in the **power** member control how sharp the specular effects are.

Specular highlights can create dramatic effects. Drawing again on the blue crystal analogy: a larger **power** value creates sharper specular highlights, making the crystal appear quite shiny. Smaller values increase the area of the effect, creating a dull reflection that make the crystal look frosty. To make an object truly matte, set the **power** member to zero and the color in **specular** to black. Experiment with different levels of reflection to produce a realistic appearance for your needs. The following illustration shows two identical models. The one on the left uses a specular reflection power of 10; the model on the right has no specular reflection.



Using Materials

This section contains information about using materials in a Microsoft® Direct3D® application. Information is divided into the following topics.

- Setting Material Properties
- Retrieving Material Properties

Setting Material Properties

Microsoft® Direct3D® rendering devices can render with one set of material properties at a time.

[C++]

In a C++ application, you set the material properties that the system uses by preparing a **D3DMATERIAL8** structure, and then calling the **IDirect3DDevice8::SetMaterial** method.

To prepare the **D3DMATERIAL8** structure for use, set the property information in the structure to create the desired effect during rendering. The following code example sets up the **D3DMATERIAL8** structure for a purple material with sharp white specular highlights.

```
D3DMATERIAL8 mat;

// Set the RGBA for diffuse reflection.
mat.Diffuse.r = 0.5f;
mat.Diffuse.g = 0.0f;
mat.Diffuse.b = 0.5f;
mat.Diffuse.a = 1.0f;

// Set the RGBA for ambient reflection.
mat.Ambient.r = 0.5f;
mat.Ambient.g = 0.0f;
mat.Ambient.b = 0.5f;
mat.Ambient.a = 1.0f;

// Set the color and sharpness of specular highlights.
mat.Specular.r = 1.0f;
mat.Specular.g = 1.0f;
mat.Specular.b = 1.0f;
mat.Specular.a = 1.0f;
mat.Power = 50.0f;
```

After preparing the **D3DMATERIAL8** structure, you apply the properties by calling the **IDirect3DDevice8::SetMaterial** method of the rendering device. This method accepts the address of a prepared **D3DMATERIAL8** structure as its only parameter. You can call **SetMaterial** with new information as needed to update the material properties for the device. The following code example shows how this might look in code.

```
// This code example uses the material properties defined for
// the mat variable earlier in this topic. The pd3dDev is assumed
// to be a valid pointer to an IDirect3DDevice8 interface.
HRESULT hr;
hr = pd3dDev->SetMaterial(&mat);
if(FAILED(hr))
{
    // Code to handle the error goes here.
}
```

[Visual Basic]
In a Microsoft® Visual Basic® application, set the material properties that the system uses by preparing a **D3DMATERIAL8** type, then calling the **Direct3DDevice8.SetMaterial** method.
To prepare the **D3DMATERIAL8** type for use, set the property information in the structure to create the desired effect during rendering. The following code example sets up the **D3DMATERIAL8** type for a purple material with sharp white specular highlights.

```
Dim mat As D3DMATERIAL8

' Set the RGBA for diffuse reflection.
mat.diffuse.r = 0.5
mat.diffuse.g = 0#
mat.diffuse.b = 0.5
mat.diffuse.a = 1#

' Set the RGBA for ambient reflection.
mat.ambient.r = 0.5
mat.ambient.g = 0#
mat.ambient.b = 0.5
mat.ambient.a = 1#

' Set the color and sharpness of specular highlights.
mat.specular.r = 1#
mat.specular.g = 1#
mat.specular.b = 1#
mat.specular.a = 1#
mat.power = 50#
```

After preparing the **D3DMATERIAL8** type, apply the properties by calling the **Direct3DDevice8.SetMaterial** method of the rendering device. This method accepts the address of a prepared **D3DMATERIAL8** type as its only parameter. You can call **SetMaterial** with new information as needed to update the material properties for the device. The following code example shows how this might look in code.

```
' This code example uses the material properties defined for
' the mat variable earlier in this topic. The d3dDevice is
' assumed to contain a valid reference to a Direct3DDevice8
' object.
On Local Error Resume Next

Call d3dDevice.SetMaterial(mat)
If Err.Number <> D3D_OK Then
    ' Code to handle the error goes here.
```

```
End If
```

When you create a Direct3D device, the current material is automatically set to the default shown in the following table.

Member	Value
Diffuse	(R:1, G:1, B:1, A:0)
Specular	(R:0, G:0, B:0, A:0)
Ambient	(R:0, G:0, B:0, A:0)
Emissive	(R:0, G:0, B:0, A:0)
Power	(0.0)

Retrieving Material Properties

[C++]
You retrieve the material properties that the rendering device is currently using by calling the **IDirect3DDevice8::GetMaterial** method for the device. Unlike the **IDirect3DDevice8::SetMaterial** method, **GetMaterial** doesn't require preparation. The **GetMaterial** method accepts the address of a **D3DMATERIAL8** structure, and fills the provided structure with information describing the current material properties before returning.

```
// For this example, the pd3dDev variable is assumed to
// be a valid pointer to an IDirect3DDevice8 interface.
HRESULT hr;
D3DMATERIAL8 mat;

hr = pd3dDev->GetMaterial(&mat);
if(FAILED(hr))
{
    // Code to handle the error goes here.
}
```

[Visual Basic]
You retrieve the material properties that the rendering device is currently using by calling the **Direct3DDevice8.GetMaterial** method for the device. Unlike the **Direct3DDevice8.SetMaterial** method, **GetMaterial** doesn't require preparation. The **GetMaterial** method accepts a variable of type **D3DMATERIAL8**, and fills it with information describing the current material properties before returning.

```
' For this example, the d3dDevice variable is assumed to
' contain a valid reference to a Direct3DDevice8 object.
On Local Error Resume Next
Dim mat As D3DMATERIAL8
```

```
Call d3dDevice.GetMaterial(mat)
If Err.Number <> D3D_OK Then
   ' Code to handle the error goes here.
End If
```

Vertex Formats

This section describes the concepts you need to understand to specify vertices in Microsoft® Direct3D®, and provides information about the various formats your application can use to declare vertices. The following topics are discussed.

- About Vertex Formats
- Untransformed and Unlit Vertices
- Untransformed and Lit Vertices
- Transformed and Lit Vertices
- Transformed and Lit Vertex Functionality
- Flexible Vertex Formats and Vertex Shaders

About Vertex Formats

A flexible vertex format (FVF) code describes the contents of vertices stored interleaved in a single data stream. It generally specifies data to be processed by the fixed function vertex processing pipeline.

[C++]
Microsoft® Direct3D® applications can define model vertices in several different ways. Support for flexible vertex definitions, also known as *flexible vertex formats* or *flexible vertex format codes*, makes it possible for your application to use only the vertex components it needs, eliminating those components that aren't used. By using only the needed vertex components, your application can conserve memory and minimize the processing bandwidth required to render models. You describe how your vertices are formatted by using a combination of Flexible Vertex Format Flags.

[Visual Basic]
Microsoft® Direct3D® applications can define model vertices in several different ways. Support for flexible vertex definitions, also known as *flexible vertex formats* or *flexible vertex format codes*, makes it possible for your application to use only the vertex components it needs, eliminating those components that aren't used. By using only the needed vertex components, your application can conserve memory and minimize the processing bandwidth required to render models. You describe how your vertices are formatted by using a combination of Flexible Vertex Format Flags.

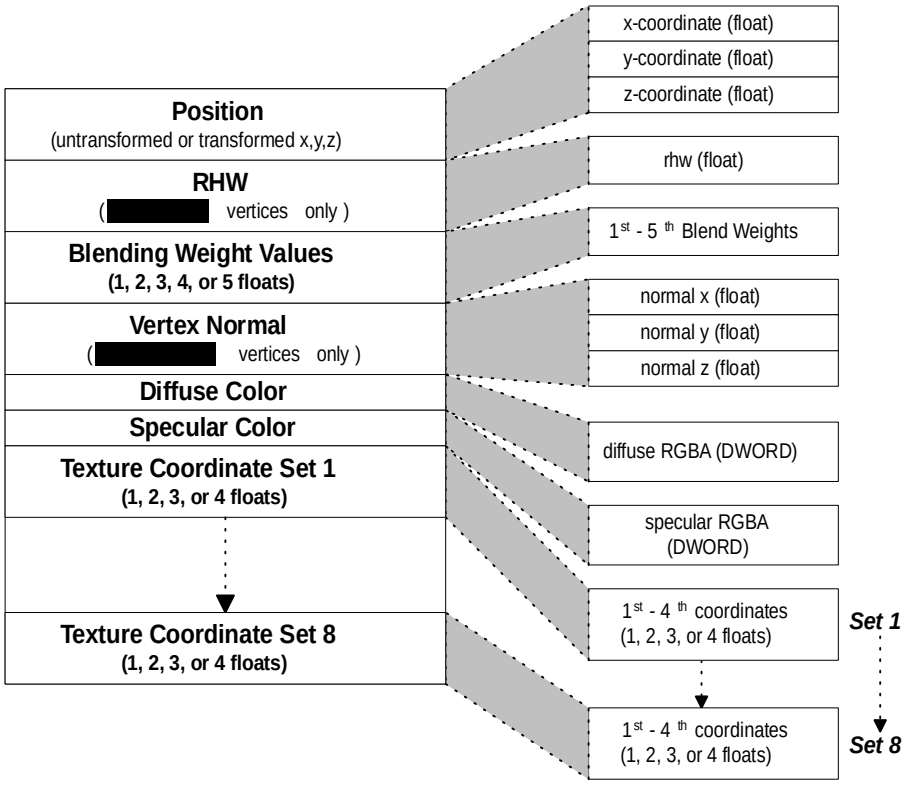
The FVF specification includes formats for point size, specified by D3DFVF_PSIZE. This size is expressed in camera space units for non-TL vertices, and in device-space units for TL vertices.

[C++]

The rendering methods of the **IDirect3DDevice8** interface provides C++ applications with methods that accept a combination of these flags, and uses them to determine how to render primitives. Basically, these flags tell the system which vertex components—position, vertex blending weights, normal, colors, the number and format of texture coordinates—your application uses and, indirectly, which parts of the rendering pipeline you want Direct3D to apply to them. In addition, the presence or absence of a particular vertex format flag communicates to the system which vertex component fields are present in memory and which you've omitted.

To determine device limitations, you can query a device for the D3DFVFCAPS_DONOTSTRIPELEMENTS and D3DFVFCAPS_TEXCOORDCOUNTMASK flexible vertex format flags. For more information, see the **FVFCaps** member of the **D3DCAPS8** structure.

One significant requirement that the system places on how you format your vertices is on the order in which the data appears. The following illustration depicts the required order for all possible vertex components in memory, and their associated data types.



Note

Texture coordinates can be declared in different formats, allowing textures to be addressed using as few as one coordinate or as many as four texture coordinates (for 2-D projected texture coordinates). For more information, see Texture Coordinate Formats. Use the **D3DFVF_TEXCOORDSIZE*n*** set of macros to create bit patterns that identify the texture coordinate formats that your vertex format uses.

No application will use every component—the reciprocal homogeneous W (RHW) and vertex normal fields are mutually exclusive. Nor will most applications try to use all eight sets of texture coordinates, but Direct3D has this capacity. There are several restrictions on which flags you can use with other flags. For example, you cannot use the D3DFVF_XYZ and D3DFVF_XYZRHW flags together, as this would indicate that your application is describing a vertex's position with both untransformed and transformed vertices.

To use indexed vertex blending, the D3DFVF_LASTBETA_UBYTE4 flag should appear at the end of the FVF. The presence of this flag indicates that the fifth blending weight will be treated as a **DWORD** instead of float. For more information, see Indexed Vertex Blending.

The following code samples shows the difference between an FVF code that uses the D3DFVF_LASTBETA_UBYTE4 flag and one that doesn't. The FVF defined below does not use the D3DFVF_LASTBETA_UBYTE4 flag. The flag D3DFVF_XYZ3 is present when four blending indices are used because you always use (1 - the sum of the first three) for the fourth.

```
#define D3DFVF_BLENDVERTEX (D3DFVF_XYZB3|D3DFVF_NORMAL|D3DFVF_TEX1)

struct BLENDVERTEX
{
    D3DXVECTOR3 v;      // Referenced as v0 in the vertex shader
    FLOAT       blend1; // Referenced as v1.x in the vertex shader
    FLOAT       blend2; // Referenced as v1.y in the vertex shader
    FLOAT       blend3; // Referenced as v1.z in the vertex shader
                        // v1.w = 1.0 - (v1.x + v1.y + v1.z)
    D3DXVECTOR3 n;      // Referenced as v3 in the vertex shader
    FLOAT       tu, tv; // Referenced as v7 in the vertex shader
};
```

The FVF defined below uses the D3DFVF_LAST_UBYTE4 flag.

```
#define D3DFVF_BLENDVERTEX (D3DFVF_XYZB4 | D3DFVF_LASTBETA_UBYTE4 |
D3DFVF_NORMAL|D3DFVF_TEX1)

struct BLENDVERTEX
{
    D3DXVECTOR3 v;      // Referenced as v0 in the vertex shader
    FLOAT       blend1; // Referenced as v1.x in the vertex shader
    FLOAT       blend2; // Referenced as v1.y in the vertex shader
    FLOAT       blend3; // Referenced as v1.z in the vertex shader
```

```
                // v1.w = 1.0 - (v1.x + v1.y + v1.z)
    DWORD       indices; // Referenced as v2.xyzw in the vertex shader
    D3DXVECTOR3 n;       // Referenced as v3 in the vertex shader
    FLOAT       tu, tv;  // Referenced as v7 in the vertex shader
};
```

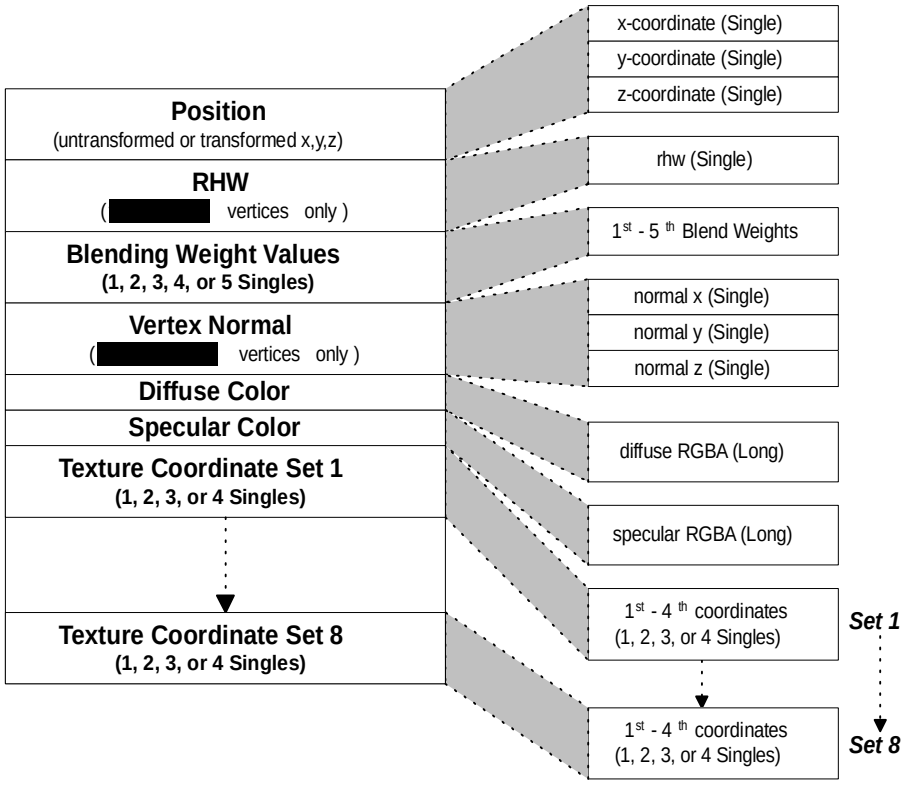
For more information, see Flexible Vertex Format Flags.

[Visual Basic]

The **Direct3DDevice8** Microsoft® Visual Basic® class includes methods that accept a combination of these flags, and uses them to determine how to render primitives. Basically, these flags tell the system which vertex components—position, vertex blending weights, normal, colors, the number and format of texture coordinates—your application uses and, indirectly, which parts of the rendering pipeline you want Direct3D to apply to them. In addition, the presence or absence of a particular vertex format flag communicates to the system which vertex component fields are present in memory and which you've omitted.

To determine device limitations, you can query a device for the D3DFVFCAPS_DONOTSTRIPELEMENTS and D3DFVFCAPS_TEXCOORDCOUNTMASK flexible vertex format flags. For more information, see the **FVFCaps** member of the **D3DCAPS8** type.

One significant requirement that the system places on how you format your vertices is on the order in which the data appears. The following illustration depicts the required order for all possible vertex components in memory, and their associated data types.



Texture coordinates can be declared in different formats, allowing textures to be addressed using as few as one coordinate, or as many as four texture coordinates for 2-D projected texture coordinates. Use the **D3DFVF_TEXCOORDSIZE*n*** set of helper functions from Math.bas to create bit patterns that identify the texture coordinate formats that your vertex format uses. The Math.bas Visual Basic code module is included with this SDK.

No real application will use every single component—the reciprocal homogeneous W (RHW) and vertex normal fields are mutually exclusive. Nor will most applications try to use all eight sets of texture coordinates, but the flexibility is there. There are several restrictions on which flags you can use with other flags. These are mostly common sense. For example, you cannot use the D3DFVF_XYZ and D3DFVF_XYZRHW flags together, as this would indicate that your application is describing a vertex's position with both untransformed and transformed vertices. For more information, take a look at the description for each of the Flexible Vertex Format Flags.

To use indexed vertex blending, the D3DFVF_LASTBETA_UBYTE4 flag should appear at the end of the FVF. The presence of this flag indicates that the fifth blending weight will be treated as a **DWORD** instead of float. For more information, see Indexed Vertex Blending.

The following code samples shows the difference between an FVF code that uses the D3DFVF_LASTBETA_UBYTE4 flag and one that doesn't. The FVF defined below

does not use the D3DFVF_LASTBETA_UBYTE4 flag. The flag D3DFVF_XYZ3 is present when four blending indices are used because you always use (1 - the sum of the first three) for the fourth.

```
Const D3DFVF_BLENDVERTEX = (D3DFVF_XYZB3 Or D3DFVF_NORMAL Or
D3DFVF_TEX1)

Private Type BLENDVERTEX
   v As D3DXVECTOR  ' Referenced as v0 in the vertex shader
   blend1 As Single ' Referenced as v1.x in the vertex shader
   blend2 As Single ' Referenced as v1.y in the vertex shader
   blend3 As Single ' Referenced as v1.z in the vertex shader
                    ' v1.w = 1.0 - (v1.x + v1.y + v1.z)
   n As D3DXVECTOR  ' Referenced as v3 in the vertex shader
   tu as Single     ' tu and tv referenced as v7 in the vertex shader
   tv As Single     '
End Type
```

The FVF defined below uses the D3DFVF_LAST_UBYTE4 flag.

```
Const D3DFVF_BLENDVERTEX = (D3DFVF_XYZB4 Or D3DFVF_LASTBETA_UBYTE4 Or _
                    D3DFVF_NORMAL Or D3DFVF_TEX1)
Private Type BLENDVERTEX
   v As D3DXVECTOR    'Referenced as v0 in the vertex shader
   blend1 As Single   ' Referenced as v1.x in the vertex shader
   blend2 As Single   ' Referenced as v1.y in the vertex shader
   blend3 As Single   ' Referenced as v1.z in the vertex shader
                      ' v1.w = 1.0 - (v1.x + v1.y + v1.z)
   indices As Long    ' Referenced as v2.xyzw in the vertex shader
   n As D3DXVECTOR    ' Referenced as v3 in the vertex shader
   tu As Single       ' tu and tv referenced as v7 in the vertex shader
   tv As Single
End Type
```

For more information, see Flexible Vertex Format Flags.

Untransformed and Unlit Vertices

[C++]

The presence of the D3DFVF_XYZ, or any D3DFVF_XYZB*n* flag, and D3DFVF_NORMAL flags in the vertex description that you pass to rendering methods identifies the untransformed and unlit vertex type. By using untransformed and unlit vertices, your application effectively requests that Microsoft® Direct3D® perform all transformation and lighting operations using its internal algorithms. You can disable the Direct3D lighting engine for the primitives being rendered by setting the D3DRS_LIGHTING render state to FALSE.

[Visual Basic]
The presence of the D3DFVF_XYZ, or any D3DFVF_XYZB*n* flag, and D3DFVF_NORMAL flags in the vertex description that you pass to rendering methods identifies the untransformed and unlit vertex type. By using untransformed and unlit vertices, your application effectively requests that Microsoft® Direct3D perform all transformation and lighting operations using its internal algorithms. You can disable the Direct3D lighting engine for the primitives being rendered by setting the D3DRS_LIGHTING render state to False.

Many applications use this vertex type, as it frees them from implementing their own transformation and lighting engines. However, because the system is making calculations for you, it requires that you provide a certain amount of information with each vertex.

- You are required to specify vertices in untransformed model coordinates. The system then applies world, view, and projection transformations to the model coordinates to position them in your scene and determine their final locations on the screen.
- You can include a vertex normal for more realistic lighting effects. Vertices lit by the system that don't include a vertex normal will use a dot product of 0 in all lighting calculations. These vertices are assumed to receive no incident light. The system uses the vertex normal, along with the current material, in its lighting calculations. For details, see Face and Vertex Normal Vectors, Lights, Materials, and Mathematics of Direct3D Lighting.

[C++]
Other than these requirements, you have the flexibility to use or disregard the other vertex components. For example, you can include a diffuse or specular color with your untransformed vertices. This was not possible before Microsoft DirectX® 6.0. Including individual colors for each vertex makes it possible to achieve shading effects that are more subtle and flexible than lighting calculations that use only the material color. Keep in mind that you must enable per-vertex color through the D3DRS_COLORVERTEX render state. Untransformed, unlit vertices can also include up to eight sets of texture coordinates.
When you define your own vertex format, remember which vertex components your application needs, and make sure they appear in the required order by declaring a properly ordered structure. The following code example declares a valid vertex format structure that includes a position, a vertex normal, a diffuse color, and two sets of texture coordinates.

```
//
// The vertex format description for this vertex would be:
// (D3DFVF_XYZ | D3DFVF_NORMAL |  D3DFVF_DIFFUSE | D3DFVF_TEX2)
//
typedef struct _UNLITVERTEX
```

```
{
    float x, y, z;     // position
    float nx, ny, nz;  // normal
    DWORD Diffuse;     // diffuse color
    float tu1, tv1;    // texture coordinates
    float tu2, tv2;
  tv2;
} UNLITVERTEX, *LPUNLITVERTEX;
```

The vertex description for the preceding structure is a combination of the D3DFVF_XYZ, D3DFVF_NORMAL, D3DFVF_DIFFUSE, and D3DFVF_TEX2 flexible vertex format flags.
For more information, see About Vertex Formats.

[Visual Basic]
Other than these requirements, you have the flexibility to use or disregard the other vertex components. For example, you can include a diffuse or specular color with your untransformed vertices. This was not possible before Microsoft DirectX® 6.0. Including individual colors for each vertex makes it possible to achieve shading effects that are more subtle and flexible than lighting calculations that use only the material color. Keep in mind that you must enable per-vertex color through the D3DRS_COLORVERTEX render state. Untransformed, unlit vertices can also include up to eight sets of texture coordinates. Microsoft Visual Basic® applications can use either the **D3DVERTEX** or **D3DVERTEX2** type for vertices.
If these types do not suit your application's needs, feel free to define your own. Remember which vertex components your application needs, and make sure they appear in the required order by declaring a properly ordered structure. The following code example declares a valid vertex format structure that includes a position, a vertex normal, a diffuse color, and two sets of texture coordinates.

```
'
' The vertex format description for this vertex would be:
' (D3DFVF_XYZ Or D3DFVF_NORMAL Or D3DFVF_DIFFUSE Or D3DFVF_TEX2)
'
Type UNLITVERTEX
    x As Single      ' Position
    y As Single
    z As Single
    nx As Single     ' Normal
    ny As Single
    nz As Single
    Diffuse As Long  ' diffuse color
    tu1 As Single    ' texture coordinates
    tv1 As Single
    tu2 As Single    ' texture coordinates
    tv2 As Single
End Type
```

The vertex description for the preceding structure would be a combination of the D3DFVF_XYZ, D3DFVF_NORMAL, D3DFVF_DIFFUSE, and D3DFVF_TEX2 flexible vertex format flags.
For more information, see About Vertex Formats.

Untransformed and Lit Vertices

[C++]
If you include the D3DFVF_XYZ flag, but not the D3DFVF_NORMAL flag, in the vertex format description you use with the Microsoft® Direct3D® rendering methods, you are identifying your vertices as untransformed but already lit. For information about other dependencies and exclusions, see the description for the Flexible Vertex Format Flags.

[Visual Basic]
If you include the D3DFVF_XYZ flag, but not the D3DFVF_NORMAL flag, in the vertex format description you use with the Microsoft® Direct3D® rendering methods, you are identifying your vertices as untransformed but already lit. For information about other dependencies and exclusions, see the description for the Flexible Vertex Format Flags.

By using untransformed and lit vertices, your application requests that Direct3D not perform any lighting calculations on your vertices, but it should still transform them using the previously set world, view, and projection matrices. Because the system isn't doing lighting calculations, it doesn't need a vertex normal. The system uses the diffuse and specular components at each vertex for shading. These colors might be arbitrary, or they might be computed using your own lighting formulas. If you don't include a diffuse or specular component, the system uses the default colors. The default value for the diffuse color is 0xFFFFFFFF, and the default value for the specular color is 0x0.
Like the other vertex types, except for including a position and some amount of color information, you are free to include or disregard the texture coordinate sets in the unlit vertex format.

[C++]
When you define your own vertex format, remember which vertex components your application needs, and make sure they appear in the required order by declaring a properly ordered structure. The following code example declares a valid untransformed and lit vertex, with diffuse and specular vertex colors, and three sets of texture coordinates.

```
//
// The vertex format description for this vertex would be:
// (D3DFVF_XYZ | D3DFVF_DIFFUSE |  D3DFVF_SPECULAR | D3DFVF_TEX3)
```

```
//
typedef struct _LITVERTEX {
    float x, y, z;   // position
    DWORD Diffuse;   // diffuse color
    DWORD Specular;  // specular color
    float tu1, tv1;  // texture coordinates
    float tu2, tv2;
    float tu3, tv3;
} LITVERTEX, *LPLITVERTEX;
```

The vertex description for the preceding structure is a combination of the D3DFVF_XYZ, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX3 flexible vertex format flags.
For more information, see About Vertex Formats.

[Visual Basic]
Applications written in Microsoft Visual Basic® can define their own vertex formats, or use either the **D3DLVERTEX** or **D3DLVERTEX2** type for lit vertices. If these types do not include all the fields that your application needs, you can define another type. Make sure that your vertex components appear in the required order, declaring a new type accordingly. The following code example declares a valid untransformed and lit vertex, with diffuse and specular vertex colors, and three sets of texture coordinates.

```
'
' The vertex format description for this vertex
' would be: (D3DFVF_XYZ Or D3DFVF_DIFFUSE Or
'    D3DFVF_SPECULAR Or D3DFVF_TEX3)
'
Type LITVERTEX
    x As Single      ' position
    y As Single
    z As Single
    Diffuse As Long  ' diffuse color
    Specular As Long ' specular color
    tu1 As Single ' texture coordinates
    tv1 As Single
    tu2 As Single
    tv2 As Single
    tu3 As Single
    tv3 As Single
End Type
```

The vertex description for the preceding type is a combination of the D3DFVF_XYZ, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX3 flexible vertex format flags.

For more information, see About Vertex Formats.

Transformed and Lit Vertices

[C++]
If you include the D3DFVF_XYZRHW flag in your vertex format description, you are telling the system that your application is using transformed and lit vertices. This means that Microsoft® Direct3D® doesn't transform your vertices with the world, view, or projection matrices, nor does it perform any lighting calculations. It assumes that your application has taken care of these steps. This fact makes transformed and lit vertices common when porting existing 3-D applications to Direct3D. In short, Direct3D does not modify transformed and lit vertices at all. It passes them directly to the driver to be rasterized.
The vertex format flags associated with untransformed vertices and lighting (D3DFVF_XYZ and D3DFVF_NORMAL) are not allowed if D3DFVF_XYZRHW is present. For more about flag dependencies and exclusions, see the description for each of the Flexible Vertex Format Flags.

[Visual Basic]
If you include the D3DFVF_XYZRHW flag in your vertex format description, you are telling the system that your application is using transformed and lit vertices. This means that Microsoft® Direct3D® doesn't transform your vertices with the world, view, or projection matrices, nor does it perform any lighting calculations. It assumes that your application has taken care of these steps. This fact makes transformed and lit vertices common when porting existing 3-D applications to Direct3D. In short, Direct3D does not modify transformed and lit vertices at all. It passes them directly to the driver to be rasterized.
The vertex format flags associated with untransformed vertices and lighting (D3DFVF_XYZ and D3DFVF_NORMAL) are not allowed if D3DFVF_XYZRHW is present. For more about flag dependencies and exclusions, see the description for each of the Flexible Vertex Format Flags.

The system requires that the vertex position you specify be already transformed. The x and y values must be in screen coordinates, and z must be the depth value of the pixel to be used in the z-buffer. Z values can range from 0.0 to 1.0, where 0.0 is the closest possible position to the user, and 1.0 is the farthest position still visible within the viewing area. Immediately following the position, transformed and lit vertices must include a reciprocal of homogeneous W (RHW) value. RHW is the reciprocal of the W coordinate from the homogeneous point (x,y,z,w) at which the vertex exists in *projection space*. This value often works out to be the distance from the eyepoint to the vertex, taken along the z-axis.
Other than the position and RHW requirements, this vertex format is similar to an untransformed and lit vertex. To recap:

- The system doesn't do any lighting calculations with this format, so it doesn't need a vertex normal.
- You can specify a diffuse or specular color. If you don't, the system uses 0x0 for specular color and 0xFFFFFFFF for diffuse color.
- You can use up to eight sets of texture coordinates, or none at all.

[C++]

When you define your own vertex format, remember which vertex components your application needs, and make sure they appear in the required order by declaring a properly ordered structure. The following code example declares a valid transformed and lit vertex, with diffuse and specular vertex colors, and one set of texture coordinates.

```
//
// The vertex format description for this vertex would be
// (D3DFVF_XYZRHW | D3DFVF_DIFFUSE |  D3DFVF_SPECULAR | D3DFVF_TEX1)
//
typedef struct _TRANSLITVERTEX {
    float x, y;      // screen position
    float z;         // Z-buffer depth
    float rhw;       // reciprocal homogeneous W
    DWORD Diffuse;   // diffuse color
    DWORD Specular;  // specular color
    float tu1, tv1;  // texture coordinates
} TRANSLITVERTEX, *LPTRANSLITVERTEX;
```

The vertex description for the preceding structure would be a combination of the D3DFVF_XYZRHW, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX1 flexible vertex format flags.
For more information, see About Vertex Formats.

[Visual Basic]

Microsoft Visual Basic® applications can use either the **D3DTLVERTEX** or **D3DTLVERTEX2** type for transformed and lit vertices. If these types do not include all the fields your application needs, you can define another type. Make sure that your vertex components appear in the required order, declaring a new type accordingly. The following code example declares a valid transformed and lit vertex, with diffuse and specular vertex colors, and one set of texture coordinates.

```
'
' The vertex format description for this vertex would be
' (D3DFVF_XYZRHW Or D3DFVF_DIFFUSE Or D3DFVF_SPECULAR Or D3DFVF_TEX1)
'
Type TRANSLITVERTEX
    x As Single      ' screen position
    y As Single
    z As Single      ' Z-buffer depth
```

```
    rhw As Single     ' reciprocal homogeneous W
    Diffuse As Long   ' diffuse color
    Specular As Long  ' specular color
    tu1 As Single     ' texture coordinates
    tv1 As Single
End Type
```

The vertex description for the preceding type is a combination of the D3DFVF_XYZRHW, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX1 flexible vertex format flags.
For more information, see About Vertex Formats and Transformed and Lit Vertex Functionality.

Transformed and Lit Vertex Functionality

Vertex data specified with a flexible vertex format (FVF) code has the property of being either transformed or nontransformed. The terminology used for FVF-specified transformed vertices is TL (transformed and lit) vertices. Microsoft® Direct3D® for Microsoft DirectX® 8.0 continues to support TL vertex data, as in previous releases. However, TL vertices are subject to some conditions that are not applicable to nontransformed vertices.
TL vertex data is possible only for vertices specified by an FVF code, and it is not valid as input for a programmable vertex shader. Programmable vertex shaders have the flexibility to take various forms of transformed vertex data, but this is different from FVF-TL vertex data because the interpretation is part of the shader, not a property of the vertex data. Because it is defined by an FVF code, TL vertex data is also inherently single stream.
TL vertex data is not guaranteed to be clipped by Direct3D, thus it is required that TL vertex primitives be clipped prior to sending them to Direct3D for rendering. The x and y values are required to be clipped to the viewport, or, if available, the device guardband. The z values for TL vertices are required to be between 0. and 1, inclusive. It is also required that the RHW values be within the range [0 < 1/**D3DCAPS8.MaxVertexW**]. The RHW requirement is to guarantee that the w values are within the supported range of the hardware rasterization device. Note that the w, and RHW, range of vertices resulting from a perspective projection transformation can be modified by applying a scale to all values in the projection matrix.

Note

If D3DPMISCCAPS_CLIPTLVERTS is set, then the device clips post-transformed vertex data to the z and (x, y) viewport limits. Clipping to user-defined clip planes is not supported for post-transformed vertex data. If the D3DPMISCCAPS_CLIPTLVERTS capability is not set, then the application is required to clip these primitives to the z limit and at least to the guardband extent in x and y. On the other hand, clipping for pre-transformed vertices is fully supported in both drawing and vertex processing calls to a destination vertex

buffer followed by another drawing call. This includes user-defined clip planes as well as the z and (x, y) viewport limits.

TL vertex data is always passed directly to the driver for rendering. When using vertex buffers with TL vertex data, there can be significant performance advantages to having the driver allocate these vertex buffers in AGP or video memory. Note that the allocation of a TL vertex buffer may be driver-allocated even when non-TL vertex data —either FVF or non-FVF—is not allowed to be driver-allocated, as would be the case when running on a hardware device that does not support transformation and lighting.
For more information, see Transformed and Lit Vertices.

Flexible Vertex Formats and Vertex Shaders

Microsoft® Direct3D® for Microsoft DirectX® 8.0 has a simplified programming model for using the fixed function vertex processing pipeline with a single input stream, providing functionality very similar to that of previous DirectX releases. In this case, the vertex shader consists of a flexible vertex format (FVF) code that is passed in place of a vertex shader handle when setting the current vertex shader. The handle space for vertex shaders is managed by the run-time library so that handles that are valid FVF codes are reserved for this usage.
Setting an FVF code as the current vertex shader causes the vertex processing to load from stream zero only, and to interpret the vertex elements as defined in the FVF code.

[C++]
The following code example illustrates how to use an FVF code as a vertex shader in your C++ application.
First, define a structure for your custom vertex type and initialize the vertices. In this case, three vertices are initialized for rendering a triangle.

```
// This code example assumes that d3dDevice is a
// valid pointer to an IDirect3DDevice8 interface.

struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex
    DWORD color;        // The vertex color
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)

CUSTOMVERTEX g_Vertices[] =
{
    { 150.0f,  50.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color
    { 250.0f, 250.0f, 0.5f, 1.0f, 0xff00ff00, },
    {  50.0f, 250.0f, 0.5f, 1.0f, 0xff00ffff, },
};
```

Then, render the primitive using stream zero.

```
d3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
d3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
```

For more information, see Vertex Shaders.

[Visual Basic]

The following code example demonstrates how to use an FVF code as a vertex shader in your Microsoft Visual Basic® application.

First, define a structure for your custom vertex type and initialize the vertices. In this case, three vertices are initialized for rendering a triangle.

```
' This code example assumes that d3dDeviceis a valid reference
' to a Direct3DDevice8 object.

Private Type CUSTOMVERTEX
    x As Single         'x in screen space
    y As Single         'y in screen space
    z  As Single        'Normalized z
    rhw As Single       'Normalized z rhw
    color As Long       'Vertex color
End Type

Const D3DFVF_CUSTOMVERTEX = (D3DFVF_XYZRHW Or D3DFVF_DIFFUSE)

Dim Vertices(2) As CUSTOMVERTEX
Dim VertexSizeInBytes As Long

VertexSizeInBytes = Len(Vertices(0))

With Vertices(0): .x = 150: .y =  50: .z = 0.5: .rhw = 1: .color = &HFFFF0000: End With
With Vertices(1): .x = 250: .y = 250: .z = 0.5: .rhw = 1: .color = &HFF00FF00: End With
With Vertices(2): .x =  50: .y = 250: .z = 0.5: .rhw = 1: .color = &HFF00FFFF: End With
```

Then, render the primitive using stream zero.

```
sizeOfVertex = Len(v)
d3dDevice.SetStreamSource 0, g_VB, sizeOfVertex
d3dDevice.SetVertexShader D3DFVF_CUSTOMVERTEX
d3dDevice.DrawPrimitive D3DPT_TRIANGLELIST, 0, 1
```

For more information, see Vertex Shaders.

Textures

Textures are a powerful tool in creating realism in computer-generated 3-D images. Microsoft® Direct3D® supports an extensive texturing feature set, providing developers with easy access to advanced texturing techniques. This section discusses the purposes and uses of textures in Direct3D. The information is presented in the following topics.

- Basic Texturing Concepts
- Texture Coordinates
- Texture Resources
- Texture Filtering
- Texture Wrapping
- Texture Blending
- Compressed Texture Resources
- Volume Texture Resources
- Automatic Texture Management
- Hardware Considerations for Texturing

Basic Texturing Concepts

This section presents the most fundamental concepts required for an understanding of texturing in Microsoft® Direct3D®. The information in this section is presented in the following topics.

- What Is a Texture?
- Texture Addressing Modes
- Texture Dirty Regions
- Texture Palettes

What Is a Texture?

Early computer-generated 3-D images, although generally advanced for their time, tended to have a shiny plastic look. They lacked the types of markings—such as scuffs, cracks, fingerprints, and smudges—that give 3-D objects realistic visual complexity. In recent years, textures have gained popularity among developers as a tool for enhancing the realism of computer-generated 3-D images.

In its everyday use, the word *texture* refers to an object's smoothness or roughness. In computer graphics, however, a texture is a bitmap of pixel colors that give an object the appearance of texture.

Because Microsoft® Direct3D® textures are bitmaps, any bitmap can be applied to a Direct3D primitive. For instance, applications can create and manipulate objects that appear to have a wood grain pattern in them. Grass, dirt, and rocks can be applied to a set of 3-D primitives that form a hill. The result is a realistic-looking hillside. You can also use texturing to create effects such as signs along a roadside, rock strata in a cliff, or the appearance of marble on a floor.

In addition, Direct3D supports more advanced texturing techniques such as texture blending—with or without transparency—and light mapping. For more information, see Texture Blending and Light Mapping with Textures.
If your application creates a hardware abstraction layer (HAL) device or a software device (see Device Types), it can use 8-, 16-, 24-, or 32-bit textures.

Texture Addressing Modes

This section describes the purpose and use of Microsoft® Direct3D® texture addressing modes. It is organized into the following topics.

- What Are Texture Addressing Modes?
- Wrap Texture Address Mode
- Mirror Texture Address Mode
- Clamp Texture Address Mode
- Border Color Texture Address Mode
- Setting and Retrieving Texture Addressing Modes
- Texture Addressing Modes and Texture Wrapping
- Device Limitations for Texture Addressing

What Are Texture Addressing Modes?

Your Microsoft® Direct3D® application can assign texture coordinates to any vertex of any primitive. For details, see Texture Coordinates. Typically, the u- and v-texture coordinates that you assign to a vertex are in the range of 0.0 to 1.0 inclusive. However, by assigning texture coordinates outside that range, you can create certain special texturing effects.
You control what Direct3D does with texture coordinates that are outside the [0.0, 1.0] range by setting the texture addressing mode. For instance, you can have your application set the texture addressing mode so that a texture is tiled across a primitive. The following topics contain additional details.

- Wrap Texture Address Mode
- Mirror Texture Address Mode
- Clamp Texture Address Mode
- Border Color Texture Address Mode

Wrap Texture Address Mode

[C++]
The wrap texture address mode, identified by the D3DTADDRESS_WRAP member of the **D3DTEXTUREADDRESS** enumerated type, makes Microsoft® Direct3D® repeat the texture on every integer junction. Suppose, for example, your application creates a square primitive and specifies texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0). Setting the texture addressing mode to D3DTADDRESS_WRAP results in the texture being applied three times in both the u-and v-directions.

[Visual Basic]
The wrap texture address mode, identified by the D3DTADDRESS_WRAP member of the **CONST_D3DTEXTUREADDRESS** enumeration, makes Microsoft® Direct3D® repeat the texture on every integer junction. Suppose, for example, your application creates a square primitive and specifies texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0). Setting the texture addressing mode to D3DTADDRESS_WRAP results in the texture being applied three times in both the u- and v-directions.

This is illustrated in the following figure.

The effects of this texture address mode are similar to, but distinct from, those of the mirror mode. For more information, see Mirror Texture Address Mode.

Mirror Texture Address Mode

[C++]
The mirror texture address mode, identified by the D3DTADDRESS_MIRROR member of the **D3DTEXTUREADDRESS** enumerated type, causes Microsoft® Direct3D® to mirror the texture at every integer boundary. Suppose, for example, your application creates a square primitive and specifies texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0). Setting the texture addressing mode to D3DTADDRESS_MIRROR results in the texture being applied three times in both the u- and v-directions. Every other row and column that it is applied to is a mirror image of the preceding row or column.

[Visual Basic]
The mirror texture address mode, identified by the D3DTADDRESS_MIRROR member of the **CONST_D3DTEXTUREADDRESS** enumeration, causes Microsoft® Direct3D® to mirror the texture at every integer boundary. Suppose, for example, your application creates a square primitive and specifies texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0). Setting the texture addressing mode to D3DTADDRESS_MIRROR results in the texture being applied three times in both the u- and v-directions. Every other row and column that it is applied to is a mirror image of the preceding row or column.

This is illustrated in the following figure.

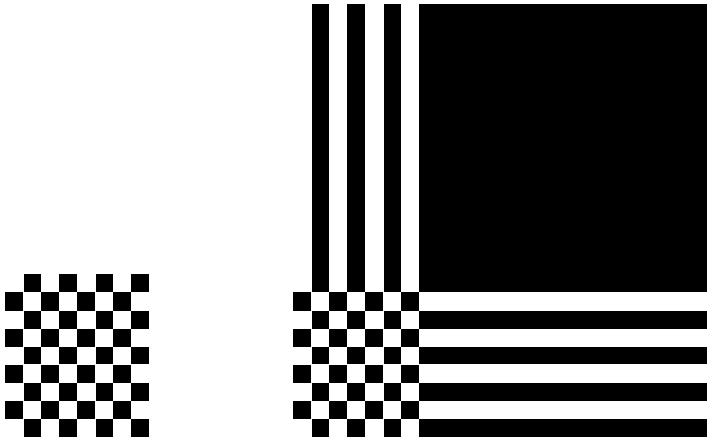
The effects of this texture address mode are similar to, but distinct from, those of the wrap mode. For more information, see Wrap Texture Address Mode.

Clamp Texture Address Mode

[C++]
The clamp texture address mode, identified by the D3DTADDRESS_CLAMP member of the **D3DTEXTUREADDRESS** enumerated type, causes Microsoft® Direct3D® to clamp your texture coordinates to the [0.0, 1.0] range. That is, it applies the texture once, then smears the color of edge pixels. For example, suppose that your application creates a square primitive and assigns texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0) to the primitive's vertices. Setting the texture addressing mode to D3DTADDRESS_CLAMP results in the texture being applied once. The pixel colors at the top of the columns and the end of the rows are extended to the top and right of the primitive respectively.

[Visual Basic]
The clamp texture address mode, identified by the D3DTADDRESS_CLAMP member of the **CONST_D3DTEXTUREADDRESS** enumeration, causes Microsoft® Direct3D® to clamp your texture coordinates to the [0.0, 1.0] range. That is, it applies the texture once, then smears the color of edge pixels. For instance, suppose that your application creates a square primitive and assigns texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0) to the primitive's vertices. Setting the texture addressing mode to D3DTADDRESS_CLAMP results in the texture being applied once. The pixel colors at the top of the columns and the end of the rows are extended to the top and right of the primitive respectively.

This is illustrated in the following figure.

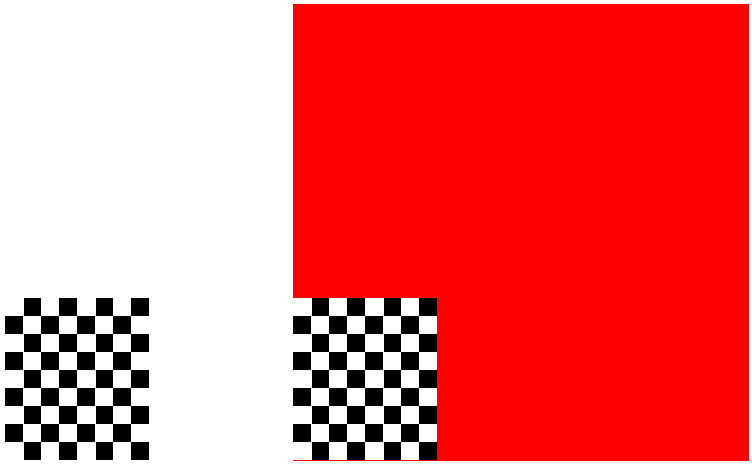


Border Color Texture Address Mode

[C++]
The border color texture address mode, identified by the D3DTADDRESS_BORDER member of the **D3DTEXTUREADDRESS** enumerated type, causes Microsoft® Direct3D® to use an arbitrary color, known as the border color, for any texture coordinates outside the range of 0.0 through 1.0, inclusive.

[Visual Basic]
The border color texture address mode, identified by the D3DTADDRESS_BORDER member of the **CONST_D3DTEXTUREADDRESS** enumeration, causes Microsoft® Direct3D® to use an arbitrary color, known as the border color, for any texture coordinates outside the range of 0.0 through 1.0, inclusive.

This is shown in the following figure in which the application specifies that the texture be applied to the primitive using a red border.



Texture

Texture with red border applied to primitive

[C++]
Applications set the border color by calling **IDirect3DDevice8::SetTextureStageState**. Set the first parameter for the call to the desired texture stage identifier, the second parameter to the D3DTSS_BORDERCOLOR stage state value, and the third parameter to the new RGBA border color.

[Visual Basic]
Microsoft Visual Basic® applications set the border color by calling **Direct3DDevice8.SetTextureStageState**. Set the first parameter for the call to the desired texture stage identifier, the second parameter to the D3DTSS_BORDERCOLOR stage state value, and the third parameter to the new RGBA border color.

Setting and Retrieving Texture Addressing Modes

[C++]
You can set texture addressing modes for individual texture stages by calling the **IDirect3DDevice8::SetTextureStageState** method. Specify the desired texture stage identifier in the first parameter. Set the second parameter to D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, or D3DTSS_ADDRESSW values to update the u-, v-, or v-addressing modes individually. The third parameter you pass to **SetTextureStageState** determines which mode is being set. This can be any member of the **D3DTEXTUREADDRESS** enumerated type. To retrieve the current texture address mode for a texture stage, call **IDirect3DDevice8::GetTextureStageState**,

using the D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, or D3DTSS_ADDRESSW members of the **D3DTEXTURESTAGESTATETYPE** enumeration to identify the address mode about which you want information.

[Visual Basic]
You can set texture addressing modes for individual texture stages by calling the **Direct3DDevice8.SetTextureStageState** method. Specify the desired texture stage identifier in the first parameter. Set the second parameter to D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, or D3DTSS_ADDRESSW values to update the u-, v-, or w-addressing modes individually. The third parameter you pass to **SetTextureStageState** determines which mode is being set. This can be any member of the **CONST_D3DTEXTUREADDRESS** enumeration.
To retrieve the current texture address mode for a given texture stage, call **Direct3DDevice8.GetTextureStageState**, using the D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, or D3DTSS_ADDRESSW members of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration to the address mode about which you want information.

Texture Addressing Modes and Texture Wrapping

Direct3D enables applications to perform texture wrapping. It is important to note that setting the texture addressing mode to D3DTADDRESS_WRAP is not the same as performing texture wrapping. Setting the texture addressing mode to D3DTADDRESS_WRAP results in multiple copies of the source texture being applied to the current primitive, and enabling texture wrapping changes how the system rasterizes textured polygons. For details, see Texture Wrapping.
Enabling texture wrapping effectively makes texture coordinates outside the [0.0, 1.0] range invalid, and the behavior for rasterizing such delinquent texture coordinates is undefined in this case. When texture wrapping is enabled, texture addressing modes are not used. Take care that your application does not specify texture coordinates lower than 0.0 or higher than 1.0 when texture wrapping is enabled.

Device Limitations for Texture Addressing

[C++]
Although the system generally allows texture coordinates outside the range of 0.0 and 1.0, inclusive, hardware limitations often affect how far outside that range texture coordinates can be. A rendering device communicates this limit in the **MaxTextureRepeat** member of the **D3DCAPS8** structure when you retrieve device capabilities. The value in this member describes the full range of texture coordinates allowed by the device. For instance, if this value is 128, then the input texture coordinates must be kept in the range -128.0 to +128.0. Passing vertices with texture coordinates outside this range is invalid. The same restriction applies to the texture coordinates generated as a result of automatic texture coordinate generation and texture coordinate transformations.

The interpretation of **MaxTextureRepeat** is also affected by the D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE capability bit. When this bit is set, the value in the **MaxTextureRepeat** member of **D3DCAPS8** is used precisely as described. However, when D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE is not set, texture repeating limitations depend on the size of the texture indexed by the texture coordinates. In this case, **MaxTextureRepeat** must be scaled by the current texture size at the largest level of detail to compute the valid texture coordinate range. For example, given a texture dimension of 32 and **MaxTextureRepeat** value of 512, the actual valid texture coordinate range is 512/32 = 16, so the texture coordinates for this device must be within the range of -16.0 to +16.0.

[Visual Basic]
Although the system generally allows texture coordinates outside the range of 0.0 and 1.0, inclusive, hardware limitations often affect how far outside that range texture coordinates can be. A rendering device communicates this limit in the **MaxTextureRepeat** member of the **D3DCAPS8** type when you retrieve device capabilities. The value in this member describes the full range of texture coordinates allowed by the device. For instance, if this value is 128, then the input texture coordinates must be kept in the range -128.0 to +128.0. Passing vertices with texture coordinates outside of this range is invalid. The same restriction applies to the texture coordinates generated as a result of automatic texture coordinate generation and texture coordinate transformations.
The interpretation of **MaxTextureRepeat** is also affected by the D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE capability bit. When this bit is set, the value in the **MaxTextureRepeat** member of **D3DCAPS8** is used precisely as described. However, when D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE is not set, texture repeating limitations depend on the size of the texture indexed by the texture coordinates. In this case, **MaxTextureRepeat** must be scaled by the current texture size at the largest level of detail to compute the valid texture coordinate range. For example, given a texture dimension of 32 and **MaxTextureRepeat** value of 512, the actual valid texture coordinate range is 512/32 = 16, so the texture coordinates for this device must be within the range of -16.0 to +16.0.

Texture Dirty Regions

[C++]
Applications can optimize what subset of a texture is copied by specifying dirty regions on textures. Only those regions marked as dirty are copied by a call to **IDirect3DDevice8::UpdateTexture**. However, the dirty regions may be expanded to optimize alignment. When a texture is created, the entire texture is considered dirty. Only the following four operations affect the dirty state of a texture.

- Adding a dirty region to a texture

- Locking some buffer in the texture. This operation adds the locked region as a dirty region. The application can turn off this automatic dirty region update if it has better knowledge of the actual dirty regions.
- Using the texture as a destination in **IDirect3DDevice8::CopyRects** marks the entire texture as dirty.
- Using the texture as a source in **UpdateTexture** clears all the dirty regions on the source texture.

[Visual Basic]
Applications can optimize what subset of a texture is copied by specifying dirty regions on textures. Only those regions marked as dirty are copies by a call to **Direct3DDevice8.UpdateTexture**. However, the dirty regions may be expanded to optimize alignment. When a texture is created, the entire texture is considered dirty. Only the following four operations affect the dirty state of a texture.

- Adding a dirty region to a texture
- Locking some buffer in the texture. This operation adds the locked region as a dirty region. The application can turn off this automatic dirty region update if it has better knowledge of the actual dirty regions.
- Using the texture as a destination in **Direct3DDevice8.CopyRects** marks the entire texture as dirty.
- Using the texture as a source in **UpdateTexture** clears all the dirty regions on the source texture.

Dirty regions are set on the top level of a mipmapped texture, and **UpdateTexture** can expand the dirty region down the mip chain in order to minimize the number of bytes copied for each sublevel. Note that the sublevel dirty region coordinates are rounded outward, that is, their fractional parts are rounded toward the nearest edge of the texture.

[C++]
Because each type of texture has different types of dirty regions, there are methods on each texture type. 2-D textures use dirty rectangle, and volume textures use boxes.

- **IDirect3DCubeTexture8::AddDirtyRect**
- **IDirect3DTexture8::AddDirtyRect**
- **IDirect3DVolumeTexture8::AddDirtyBox**

Passing NULL for the *pDirtyRect* or *pDirtyBox* parameters for the above methods expands the dirty region to cover the entire texture.

[Visual Basic]

Because each type of texture has different types of dirty regions, there are methods on each texture type. 2-D textures use dirty rectangle, and volume textures use boxes.

- **Direct3DCubeTexture8.AddDirtyRect**
- **Direct3DTexture8.AddDirtyRect**
- **Direct3DVolumeTexture8.AddDirtyBox**

Passing ByVal 0 for the *DirtyRect* or *DirtyBox* parameters for the above methods expands the dirty region to cover the entire texture.

Each lock method can take D3DLOCK_NO_DIRTY_UPDATE, which prevents any changes to the dirty state of the texture. For more information, see Locking Resources.
Applications should use D3DLOCK_NO_DIRTY_UPDATE when further information about the true set of regions changed during a lock operation is available. You should note that a lock or copy to only a sublevel—that is, without locking or copying to the top level—of texture does not update the dirty regions for that texture. Applications assume the same responsibility for updating dirty regions when they lock lower levels without locking the topmost level.

Texture Palettes

[C++]
Microsoft® Direct3D® for Microsoft DirectX® 8.0 supports paletted textures through a set of 256 entry palettes associated with the **IDirect3DDevice8** object. A palette is made current by calling the **IDirect3DDevice8::SetCurrentTexturePalette** method. The current palette is used for translating all paletted textures for all active texture stages. **IDirect3DDevice8::SetPaletteEntries** updates all of a palette's 256 entries. Each entry is a **PALETTEENTRY** structure of the format D3DFMT_A8R8G8B8. All entries default to 0xFFFFFFFF. Note that as of DirectX 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not work as documented in the Microsoft Platform Software Development Kit (SDK). The *peFlags* member is now the alpha channel for 8-bit palettized formats.
The **IDirect3DDevice8** palettes contain an alpha channel. This alpha channel can be used when the D3DPTEXTURECAPS_ALPHAPALETTE device capability flag is set, indicating that the device supports alpha from the palette. The palette alpha channel is used when the texture format does not have an alpha channel. If the device does not support alpha from the palette and the texture format does not have an alpha channel, then a value of 0xFF is used for alpha.

[Visual Basic]
Microsoft® Direct3D® for Microsoft DirectX® 8.0 supports paletted textures through a set of 256 entry palettes associated with the **Direct3DDevice8** object. A palette is made current by calling the **Direct3DDevice8.SetCurrentTexturePalette** method. The current palette is used for translating all paletted textures for all active texture stages. **Direct3DDevice8.SetPaletteEntries** updates all of a palette's 256

entries. Each entry is a **PALETTEENTRY** type of the format D3DFMT_A8R8G8B8. All entries default to 0xFFFFFFFF. Note that as of DirectX 8, the *peFlags* member of the **PALLETTEENTRY** structure does not work as documented in the Microsoft Platform Software Development Kit (SDK). The *peFlags* member is now the alpha channel for 8-bit palettized formats.
The **Direct3DDevice8** palettes contain an alpha channel. This alpha channel can be used when the D3DPTEXTURECAPS_ALPHAPALETTE device capability flag is set, indicating that the device supports alpha from the palette. The palette alpha channel is used when the texture format does not have an alpha channel. If the device does not support alpha from the palette and the texture format does not have an alpha channel, then a value of 0xFF is used for alpha.

There is a maximum of 16,384 palettes. Because memory resources associated with the set of palettes are proportional to the maximum palette number that an application references, use contiguous palette numbers starting at zero.

Texture Coordinates

The following topics introduce the concept of texture coordinates, their formats, and the Microsoft® Direct3D® services used to manipulate them.

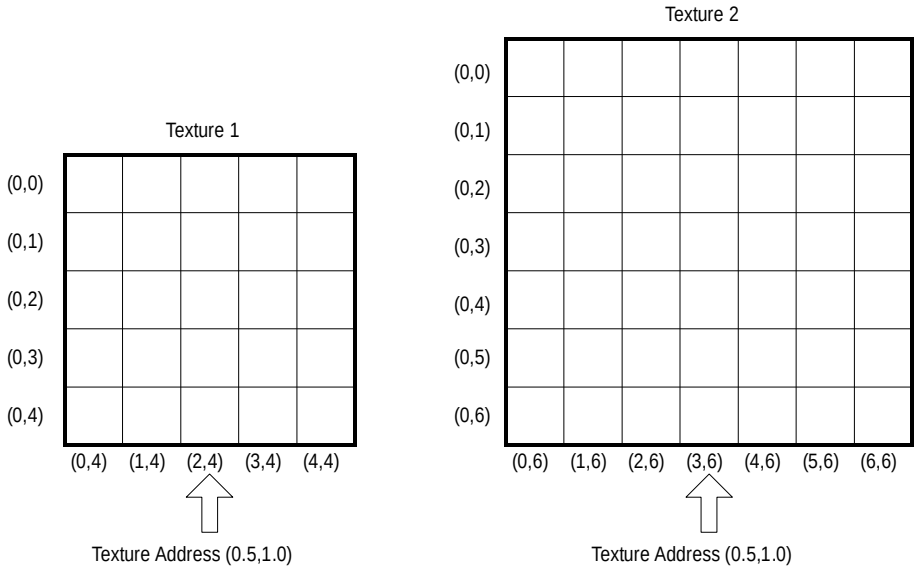
- Understanding Texture Coordinates
- Directly Mapping Texels to Pixels
- Texture Coordinate Formats
- Texture Coordinate Processing

Understanding Texture Coordinates

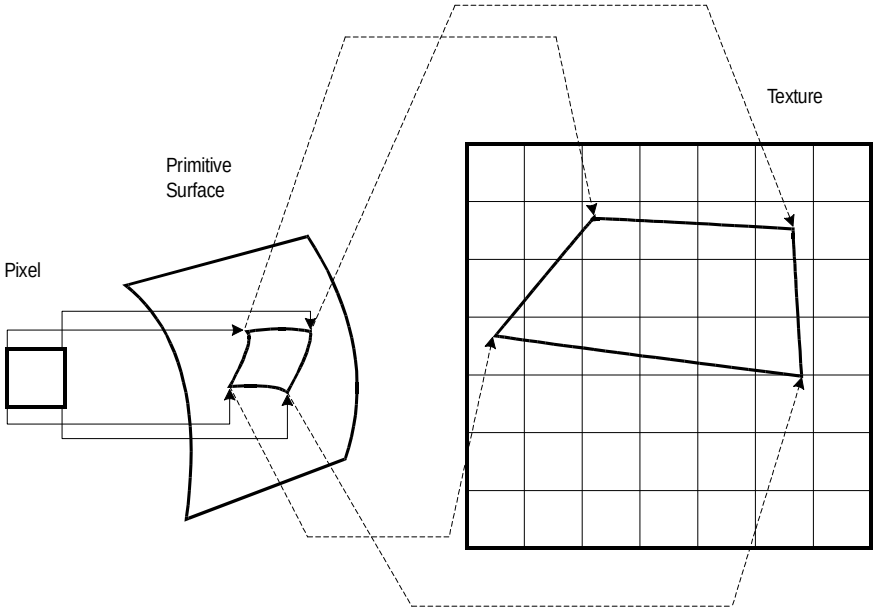
Most textures, like bitmaps, are a two-dimensional array of color values. Cubic-environment map textures are an exception. For details, see Cubic Environment Mapping. The individual color values are called a texture element, or texel. Each texel has a unique address in the texture. The address can be thought of as a column and row number, which are labeled u and v respectively.
Texture coordinates are in texture space. That is, they are relative to the location (0,0) in the texture. When a texture is applied to a primitive in 3-D space, its texel addresses must be mapped into object coordinates. They must then be translated into screen coordinates, or pixel locations.
Microsoft® Direct3D® maps texels in texture space directly to pixels in screen space, skipping the intermediate step for greater efficiency. This mapping process is actually an inverse mapping. That is, for each pixel in screen space, the corresponding texel position in texture space is calculated. The texture color at or around that point is sampled. The sampling process is called texture filtering. For more information, see Texture Filtering.
Each texel in a texture can be specified by its texel coordinate. However, in order to map texels onto primitives, Direct3D requires a uniform address range for all texels in all textures. Therefore, it uses a generic addressing scheme in which all texel

addresses are in the range of 0.0 to 1.0 inclusive. Direct3D applications specify texture coordinates in terms of u,v values, much like 2-D Cartesian coordinates are specified in terms of x,y coordinates. Technically, the system can actually process texture coordinates outside the range of 0.0 and 1.0, and does so by using the parameters you set for texture addressing. For more information, see Texture Addressing Modes.

A result of this is that identical texture addresses can map to different texel coordinates in different textures. In the following illustration, the texture address being used is (0.5,1.0). However, because the textures are different sizes, the texture address maps to different texels. Texture 1, on the left, is 5x5. The texture address (0.5,1.0) maps to texel (2,4). Texture 2, on the right, is 7x7. The texture address (0.5,1.0) maps to texel (3,6).

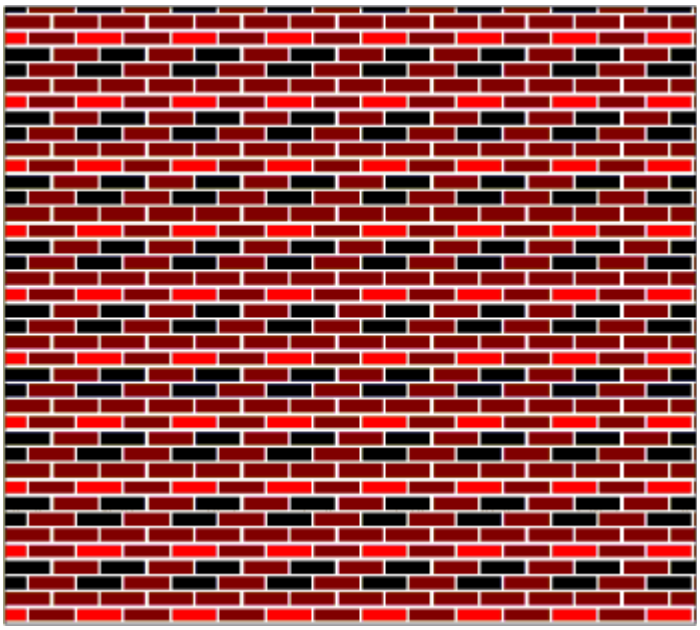


A simplified version of the texel mapping process is shown in the following diagram. Admittedly, this example is extremely simple. For more detailed information, see Directly Mapping Texels to Pixels.



For this example, a pixel, shown at the left of the illustration, is idealized into a square of color. The addresses of the four corners of the pixel are mapped onto the 3-D primitive in object space. The shape of the pixel is often distorted because of the shape of the primitive in 3-D space and because of the viewing angle. The corners of the surface area on the primitive that correspond the corners of the pixel are then mapped into texture space. The mapping process distorts the pixel's shape again, which is common. The final color value of the pixel is computed from the texels in the region to which the pixel maps. You determine the method that Direct3D uses to arrive at the pixel color when you set the texture filtering method. For more information, see Texture Filtering.

Your application can assign texture coordinates directly to vertices. This capability gives you control over which portion of a texture is mapped onto a primitive. For instance, suppose you create a rectangular primitive that is exactly the same size as the texture in the following illustration. In this example, you want your application to map the whole texture onto the whole wall. The texture coordinates your application assigns to the vertices of the primitive are (0.0,0.0), (1.0,0.0), (1.0,1.0), and (0.0,1.0).



If you decide to decrease the height of the wall by one-half, you can distort the texture to fit onto the smaller wall, or you can assign texture coordinates that cause Direct3D to use the bottom half of the texture.
If you decide to distort or scale the texture to fit the smaller wall, the texture filtering method that you use will influence the quality of the image. For more information, see Texture Filtering.
If, instead, you decide to assign texture coordinates to make Direct3D use the bottom half of the texture for the smaller wall, the texture coordinates your application assigns to the vertices of the primitive in this example are (0.0,0.0), (1.0,0.0), (1.0,0.5), and (0.0,0.5). Direct3D applies the bottom half of the texture to the wall.
It is possible for texture coordinates of a vertex to be greater than 1.0. When you assign texture coordinates to a vertex that are not in the range of 0.0 to 1.0 inclusive, you should also set the texture addressing mode. For further information, see Texture Addressing Modes.

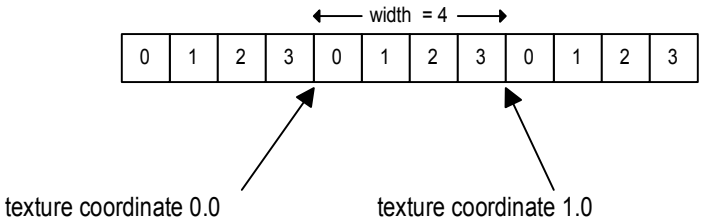
Directly Mapping Texels to Pixels

Applications often need to apply textures to geometry in a scene so that texels map directly to on-screen pixels. For example, take an application that needs to display text within a texture on an object within a scene. In order to clearly display textual information in a texture, the application needs some way to ensure that the textured geometry receives texels undisrupted by texture filtering. Failing this, the resulting image is often blurred, or in the case of point-sampled texture filtering, can cause rough edges.
Because the Microsoft® Direct3D® pixel and texture sampling rules are carefully defined to unify pixel and texture sampling while supporting image and texture filtering, getting the texels in a texture to map directly to on-screen pixels can be a

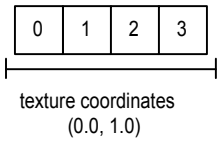
significant and often frustrating challenge. Overcoming this challenge requires a clear understanding of how Direct3D maps the floating-point texture coordinates for a vertex to the integer pixel coordinates used by the rasterizer.
Direct3D performs the following computations to map floating point texture coordinates to texel addresses.

$$T_x = (u \times M_x) - 0.5$$
$$T_y = (v \times M_y) - 0.5$$

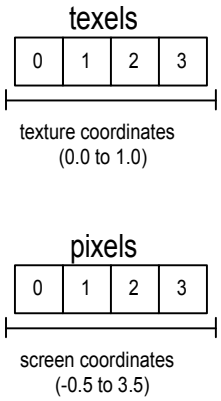
In these formulas, T_x and T_y are the horizontal and vertical output texel coordinates, and u and v are the horizontal and vertical texture coordinates supplied for the vertex. The M_x and M_y elements represent the number of horizontal or vertical texels at the current mipmap level. The remainder of this discussion targets horizontal mapping of texels to pixels; keep in mind that mapping in the vertical is identical to the horizontal case.
Placing the texture coordinate limits 0.0 and 1.0 into these formulas maps a texture coordinate of 0.0 halfway between the first and last texels of a repeated texture map. The coordinate 1.0 is mapped halfway between the last texel of the current iteration of the texture map and the next iteration. For a repeated texture that's four texels wide, at mipmap level 0, the following illustration shows where the system maps the coordinates 0.0 and 1.0.



Given an understanding of this mapping, you can apply a simple bias to your screen-space geometry coordinates to force the system to map each texel to a corresponding pixel. For example, to draw a four-sided polygon that maps each texel from the preceding texture to one, and only one, pixel on the screen, you must force geometry coordinates to overlap the pixels, effectively placing the center of each texel at the center of each pixel. The result is the 1-to-1 mapping often sought-after by applications.
To map the texture with a 4-texel width to the pixel coordinates 0 through 3, draw a four-sided polygon, from two triangles, that has screen-space coordinates of -0.5 to 3.5 and texture coordinates of 0.0 to 1.0. For example, take the pixel at screen coordinate 0.0. Because 0.0 is one-half pixel away from the first vertex, at -0.5, and the total width is 4.0, the iterated texture coordinate is 0.125. Scaling this by the texture size, which is 4, results in the coordinate 0.5. Subtracting this 0.5 bias produces a texture address of 0.0, which fully maps to the first texel in the map.
To summarize, texture coordinates overlap the texture map evenly on both sides. The following illustration shows the mapping of a texture that is four texels wide.



The system normalizes pixel coordinates in the same way it normalizes texel coordinates. Therefore, if the vertices overlap the pixels into which to render, and if the vertices use texture coordinates of 0.0 and 1.0, the pixels and texels line up. If both are of similar size and properly aligned, they match exactly, texel to pixel, as shown in the following figure.



Texture Coordinate Formats

Texture coordinates in Microsoft® Direct3D® can include one, two, three, or four floating point elements to address textures with varying levels of dimension. A 1-D texture—a texture surface with dimensions of 1-by-*n* texels—is addressed by one texture coordinate. The most common case, 2-D textures, are addressed with two texture coordinates commonly called *u* and *v*. Direct3D supports a single type of 3-D texture, called a *cubic-environment map*. Cubic environment maps aren't truly 3-D, but they are addressed with a 3-element vector. For details, see Cubic Environment Mapping.

[C++]
As described in About Vertex Formats, applications encode texture coordinates in the vertex format. The vertex format can include multiple sets of texture coordinates. Use the D3DFVF_TEX0 through D3DFVF_TEX8 flexible vertex format flags to describe a vertex format that includes no texture coordinates, or as many as eight sets.
Each texture coordinate set can have between one and four elements. The D3DFVF_TEXTUREFORMAT1 through D3DFVF_TEXTUREFORMAT4 flags describe the number of elements in a texture coordinate in a set, but these flags aren't used by themselves. Rather, the **D3DFVF_TEXCOORDSIZE*n*** set of macros use these flags to create bit patterns that describe the number of elements used by a

particular set of texture coordinates in the vertex format. These macros accept a single parameter that identifies the index of the coordinate set whose number of elements is being defined. The following example illustrates how these macros are used.

```
// This vertex format contains two sets of texture coordinates.
// The first set (index 0) has 2 elements, and the second set
// has 1 element. The description for this vertex format would be:
//     dwFVF = D3DFVF_XYZ  | D3DFVF_NORMAL | D3DFVF_DIFFUSE | D3DFVF_TEX2 |
//             D3DFVF_TEXCOORDSIZE2(0) | D3DFVF_TEXCOORDSIZE1(1);
//
typedef struct CVF
{
    D3DVECTOR position;
    D3DVECTOR normal;
    D3DCOLOR  diffuse;
    float     u, v;   // 1st set, 2-D
    float     t;      // 2nd set, 1-D
} CustomVertexFormat;
```

[Visual Basic]

As described in About Vertex Formats, applications encode texture coordinates in the vertex format. The vertex format can include multiple sets of texture coordinates. Use the D3DFVF_TEX0 through D3DFVF_TEX8 flexible vertex format flags to describe a vertex format that includes no texture coordinates, or as many as eight sets.

Each texture coordinate set can have between one and four elements. Use the D3DFVF_TEXCOORDSIZE*n* set of helper functions in the Math.bas source file that ships with this SDK to generate bit patterns that describe the number of elements used by a set of texture coordinates in the vertex format. These macros accept a single parameter that identifies the index of the coordinate set whose number of elements is being defined. The following example illustrates how these macros are used.

```
Private Sub Form_Load()
' This vertex format contains two sets of texture coordinates.
' The first set (index 0) has 2 elements, and the second set
' has 1 element.
'
' The description for this vertex format would be:
'    lFVF = D3DFVF_XYZ Or D3DFVF_NORMAL Or D3DFVF_DIFFUSE Or D3DFVF_TEX2 Or
_
'           D3DFVF_TEXCOORDSIZE2(0) Or D3DFVF_TEXCOORDSIZE1(1)
'
Type CustomVertexFormat
    position As D3DVECTOR
    normal As D3DVECTOR
    diffuse  As D3DVECTOR
    ' 1st set, 2-D
    u As Single
```

```
    v As Single
    ' 2nd set, 1-D
    t As Single
End Type
```

Note

With the exception of cubic-environment maps, rasterizers cannot address textures by using any more than two elements. Applications can supply up to three elements for a texture coordinate, but only if the texture is a cube map, or the D3DTTFF_PROJECTED texture transform flag is used. The D3DTTFF_PROJECTED flag causes the rasterizer to divide the first two elements by the third (or n^{th}) element. For more information, see Texture Coordinate Transformations.

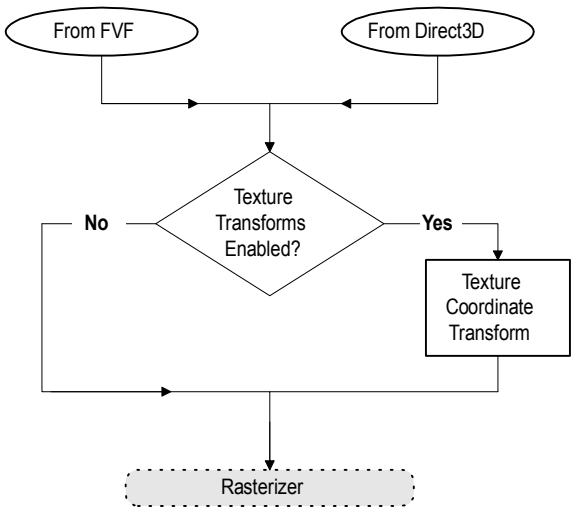
Texture Coordinate Processing

This section describes Microsoft® Direct3D® support for processing texture coordinates. Information is organized into the following topics.

- Texture Coordinate Processing Path
- Automatically Generated Texture Coordinates
- Texture Coordinate Transformations
- Using Texture Coordinate Processing

Texture Coordinate Processing Path

The following figure shows the path taken by the texture coordinates from their source, through processing, and to the rasterizer.



[C++]
There are two sources from which the system can draw texture coordinates. For a given texture stage, you can use texture coordinates included in the vertex format (D3DFVF_TEX1 through D3DFVF_TEX8), or you can use texture coordinates automatically generated by Microsoft® Direct3D®. For details about the latter case, see Automatically Generated Texture Coordinates. If the **D3DTSS_TEXTURETRANSFORMFLAGS** texture stage state for the current texture stage is set to D3DTTFF_DISABLE (the default setting), input coordinates are not transformed. If **D3DTSS_TEXTURETRANSFORMFLAGS** is set to any other value, the transformation matrix for that stage is applied to the input coordinates.
The **D3DTEXTURETRANSFORMFLAGS** enumerated type defines valid values for the **D3DTSS_TEXTURETRANSFORMFLAGS** texture-stage state. With the exception of the D3DTTFF_DISABLE flag, which bypasses texture coordinate transformation, the values defined in this enumeration configure the number of output coordinates that the system passes to the rasterizer. The D3DTTFF_COUNT1 through D3DTTFF_COUNT4 flags instruct the system to pass one, two, three, or four elements from the output coordinates to the rasterizer. Currently 1-D texture coordinates aren't supported.
The D3DTTFF_PROJECTED flag is special: it tells the system that the texture coordinates are for a projected texture. Combine the D3DTTFF_PROJECTED flag with another member of **D3DTEXTURETRANSFORMFLAGS** to instruct the rasterizer to divide all the elements by the last element before rasterization takes place. For example, when explicitly using three-element texture coordinates, or when transformation results in a three-element texture coordinate, you can combine the D3DTTFF_COUNT3 and D3DTTFF_PROJECTED flags to cause the rasterizer to divide the first two elements by the last, producing 2-D texture coordinates required to address a 2-D texture.

[Visual Basic]
There are two sources from which the system can draw texture coordinates. For a given texture stage, you can use texture coordinates included in the vertex format (D3DFVF_TEX1 through D3DFVF_TEX8), or you can use texture coordinates automatically generated by Microsoft® Direct3D®. For details about the latter case, see Automatically Generated Texture Coordinates. If the **D3DTSS_TEXTURETRANSFORMFLAGS** texture stage state for the current texture stage is set to D3DTTFF_DISABLE (the default setting), input coordinates are not transformed. If **D3DTSS_TEXTURETRANSFORMFLAGS** is set to any other value, the transformation matrix for that stage is applied to the input coordinates.
The **CONST_D3DTEXTURETRANSFORMFLAGS** enumeration defines valid values for the D3DTSS_TEXTURETRANSFORMFLAGS texture-stage state. With the exception of the D3DTTFF_DISABLE flag, which bypasses texture coordinate transformation, the values defined in this enumeration configure the number of output coordinates that the system passes to the rasterizer. The D3DTTFF_COUNT1 through

D3DTTFF_COUNT4 flags instruct the system to pass one, two, three, or four elements from the output coordinates to the rasterizer.
The D3DTTFF_PROJECTED flag is special: it tells the system that the texture coordinates are for a projected texture. Combine the D3DTTFF_PROJECTED flag with another member of **CONST_D3DTEXTURETRANSFORMFLAGS** to instruct the rasterizer to divide all the elements by the last element before rasterization takes place. For instance, when explicitly using three-element texture coordinates, or when transformation results in a three-element texture coordinate, you can combine the D3DTTFF_COUNT3 and D3DTTFF_PROJECTED flags to cause the rasterizer to divide the first two elements by the last, producing 2-D texture coordinates required to address a 2-D texture.

Note

With the exception of cubic-environment maps, rasterizers cannot address textures by using texture coordinates with more than two elements. If you specify more elements than can be used to address the current texture for that stage, the extraneous elements are ignored. This also applies when using 2-D texture coordinates for a 1-D texture.

Automatically Generated Texture Coordinates

Microsoft® Direct3D® can automatically generate the texture coordinates for a vertex. This section describes the services offered by the system and includes details on how the services are used. The following topics are discussed.

- About Automatically Generated Texture Coordinates
- Configuring Automatically Generated Texture Coordinates

About Automatically Generated Texture Coordinates

The system can use the transformed camera-space position or the normal from a vertex as texture coordinates, or it can compute the three element vectors used to address a cubic environment map. Like texture coordinates that you explicitly specify in a vertex, you can use automatically generated texture coordinates as input for texture coordinate transformations.
Automatically generated texture coordinates can significantly reduce the bandwidth required for geometry data by eliminating the need for explicit texture coordinates in the vertex format. In many cases, the texture coordinates that the system generates can be used with transformations to produce special effects. Of course, this is a special-purpose feature, and you will use explicit texture coordinates for many occasions.

Configuring Automatically Generated Texture Coordinates

[C++]
In C++, the D3DTSS_TEXCOORDINDEX texture-stage state (from the **D3DTEXTURESTAGESTATETYPE** enumerated type) controls how the system generates texture coordinates.

[Visual Basic]
In Microsoft® Visual Basic®, the D3DTSS_TEXCOORDINDEX texture-stage state (from the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration) controls how the system generates texture coordinates.

Normally, this state instructs the system to use a particular set of texture coordinates encoded in the vertex format. When you include the D3DTSS_TCI_CAMERASPACENORMAL, D3DTSS_TCI_CAMERASPACEPOSITION, or D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR flags in the value that you assign to this state, the system behavior is quite different. If any of these flags are present, the texture stage ignores the texture coordinates within the vertex format in favor of coordinates that the system generates. The meanings for each flag are shown in the following list.

D3DTSS_TCI_CAMERASPACENORMAL
: Use the vertex normal, transformed to camera space, as input texture coordinates.

D3DTSS_TCI_CAMERASPACEPOSITION
: Use the vertex position, transformed to camera space, as input texture coordinates.

D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR
: Use the reflection vector, transformed to camera space, as input texture coordinates. The reflection vector is computed from the input vertex position and normal vector.

The preceding flags are mutually exclusive. If you include one flag, you can still specify an index value, which the system uses to determine the texture wrapping mode.

[C++]
The following code example shows how these flags are used in C++.

```
/*
 * For this example, the d3dDevice variable is a valid
 * pointer to an IDirect3DDevice8 interface.
 *
 * Use the vertex position (camera-space) as the input
 * texture coordinates for this texture stage, and the
 * wrap mode set in the D3DRENDERSTATE_WRAP1 render state.
 */
d3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
                                 D3DTSS_TCI_CAMERASPACEPOSITION | 1 );
```

[Visual Basic]
The following code example shows how these flags are used in Visual Basic.

```
'
' For this example, the D3DDevice variable is a valid
```

```
' reference to a Direct3DDevice8 object.
'
' Use the vertex position (camera-space) as the input
' texture coordinates for this texture stage, and the
' wrap mode set in the D3DRENDERSTATE_WRAP1 render state.
'
Call D3DDevice.SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, _
                        D3DTSS_TCI_CAMERASPACEPOSITION Or 1 )
```

Note

Automatically generated texture coordinates are most useful as input values for a texture coordinate transformation, or to eliminate the need for your application to compute three-element vectors for cubic-environment maps.

See Also

Texture Coordinate Transformations, Cubic Environment Mapping

Texture Coordinate Transformations

The following topics provide information about working with the texture coordinate transformation features offered by Microsoft® Direct3D® Immediate Mode.

- About Texture Coordinate Transformations
- Setting and Retrieving Texture Coordinate Transformations
- Enabling Texture Coordinate Transformations

About Texture Coordinate Transformations

Microsoft® Direct3D® devices can transform the texture coordinates for vertices by applying a 4×4 matrix. The system applies transformations to texture coordinates in the same manner as geometry, and any transformations that can be communicated in a 4×4 matrix—scales, rotations, translations, projections, shears, or any combination of these—can be applied.

Note

Direct3D does not modify transformed and lit vertices. As a result, an application using transformed and lit vertices cannot use Direct3D to transform the texture coordinates of the vertices.

Devices that support hardware-accelerated transformation and lighting operations (TnLHAL Device) also accelerate the transformation of texture coordinates. When hardware acceleration of transformations isn't available, platform-specific optimizations in the Direct3D geometry pipeline apply to texture coordinate transformations.

Texture coordinate transformations are useful for producing special effects while avoiding the need to directly modify the texture coordinates of your geometry. You could use simple translation or rotation matrices to animate textures on an object, or

you can transform texture coordinates that are automatically generated by Direct3D to simplify and perhaps accelerate advanced effects such as projected textures and dynamic light-mapping. Additionally, you might use texture coordinate transforms to reuse a single set of texture coordinates for multiple purposes, in multiple texture stages.
For more information, see Using Texture Coordinate Processing.

Setting and Retrieving Texture Coordinate Transformations

[C++]
Like the matrices that your application uses for geometry, you set and retrieve texture coordinate transformations by calling the **IDirect3DDevice8::SetTransform** and **IDirect3DDevice8::GetTransform** methods. These methods accept the D3DTS_TEXTURE0 through D3DTS_TEXTURE7 members of the **D3DTRANSFORMSTATETYPE** enumerated type to identify the transformation matrices for texture stages 0 through 7, respectively.
The following code sets a matrix to apply to the texture coordinates for texture stage 0.

```
// For this example, the d3dDevice variable contains a
// valid pointer to an IDirect3DDevice8 interface.
//
D3DMATRIX matTrans = D3DXMatrixIdentity( NULL );

// Set-up the matrix for the desired transformation.
d3dDevice->SetTransform( D3DTS_TEXTURE0, &matTrans );
```

[Visual Basic]
Like the matrices that your application uses for geometry, you set and retrieve texture coordinate transformations by calling the **Direct3DDevice8.SetTransform** and **Direct3DDevice8.GetTransform** methods. These methods accept the D3DTS_TEXTURE0 through D3DTS_TEXTURE7 members of the **CONST_D3DTRANSFORMSTATETYPE** enumeration to identify the transformation matrices for texture stages 0 through 7, respectively.
The following code sets a matrix to apply to the texture coordinates for texture stage 0.

```
' For this example, the D3DDevice variable contains a
' valid reference to a Direct3DDevice8 object, and
' dx contains a valid reference to a DirectX8 object.

Dim matTrans As D3DMATRIX

Call D3DXMatrixIdentity( matTrans )

' Set-up the matrix for the desired transformation.
D3dDevice.SetTransform( D3DTS_TEXTURE0, matTrans )
```

Enabling Texture Coordinate Transformations

[C++]
The **D3DTSS_TEXTURETRANSFORMFLAGS** texture stage state controls the application of texture coordinate transformations. Values for this texture stage state are defined by the **D3DTEXTURETRANSFORMFLAGS** enumerated type. Texture coordinate transformations are disabled when D3DTSS_TEXTURETRANSFORMFLAGS is set to D3DTTFF_DISABLE (the default value). Assuming that texture coordinate transformations were enabled for stage 0, the following code disables them.

```
// For this example, the d3dDevice variable contains a
// valid pointer to an IDirect3DDevice8 interface.

D3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTFF_DISABLE );
```

The other values defined in **D3DTEXTURETRANSFORMFLAGS** are used to enable texture coordinate transformations, and to control how many resulting texture coordinate elements are passed to the rasterizer. For example, take the following code.

```
// For this example, the d3dDevice variable contains a
// valid pointer to an IDirect3DDevice8 interface.

D3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTFF_COUNT2 );
```

The D3DTTFF_COUNT2 value instructs the system to apply the transformation matrix set for texture stage 0, and then pass the first two elements of the modified texture coordinates to the rasterizer.
The D3DTTFF_PROJECTED texture transformation flag indicates coordinates for a projected texture. When this flag is specified, the rasterizer divides the elements passed-in by the last element. Take the following code, for example.

```
// For this example, the d3dDevice variable contains a
// valid pointer to an IDirect3DDevice8 interface.

d3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
                    D3DTTFF_COUNT3 | D3DTTFF_PROJECTED );
```

This example informs the system to pass three texture coordinate elements to the rasterizer. The rasterizer divides the first two elements by the third, producing the 2-D texture coordinates needed to address the texture.

[Visual Basic]
The D3DTSS_TEXTURETRANSFORMFLAGS texture stage state controls the application of texture coordinate transformations. Values for this texture stage state are defined by the **CONST_D3DTEXTURETRANSFORMFLAGS** enumeration.

Texture coordinate transformations are disabled when D3DTSS_TEXTURETRANSFORMFLAGS is set to D3DTTFF_DISABLE (the default value). Assuming that texture coordinate transformations were enabled for stage 0, the following code example disables them.

```
' For this example, the D3DDevice variable contains a
' valid reference to a Direct3DDevice8 object.

Call D3DDevice.SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, _
                    D3DTTFF_DISABLE)
```

The other values defined in **CONST_D3DTEXTURETRANSFORMFLAGS** are used to enable texture coordinate transformations, and to control how many resulting texture coordinate elements are passed to the rasterizer. For example, take the following code.

```
' For this example, the D3DDevice variable contains a
' valid reference to a Direct3DDevice8 object.

Call D3DDevice.SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, _
                    D3DTTFF_COUNT2)
```

The D3DTTFF_COUNT2 value instructs the system to apply the transformation matrix set for texture stage 0, and then pass the first two elements of the modified texture coordinates to the rasterizer.
The D3DTTFF_PROJECTED texture transformation flag indicates coordinates for a projected texture. When this flag is specified, the rasterizer divides the elements passed-in by the last element. Take the following code, for example.

```
' For this example, the D3DDevice variable contains a
' valid pointer to an IDirect3DDevice8 interface.

Call D3DDevice.SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, _
                    D3DTTFF_COUNT3 | D3DTTFF_PROJECTED)
```

This example informs the system to pass three texture coordinate elements to the rasterizer. The rasterizer divides the first two elements by the third, producing the 2-D texture coordinates needed to address the texture.

Using Texture Coordinate Processing

[C++]
The following list contains examples of ways you might use texture coordinate processing to achieve special texturing effects.

Animating textures (by translation or rotation) on a model

- Define 2-D texture coordinates in your vertex format.

```
// Use a single texture, with 2-D texture coordinates. This
// bit-pattern should be expanded to include position, normal,
```

```
// and color information as needed.
DWORD dwFVFTex = D3FVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0);
```

- Configure the rasterizer to use 2-D texture coordinates.

```
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTFF_COUNT2);
```

- Define and set an appropriate texture coordinate transformation matrix.

```
// M is a D3DMATRIX being set to translate texture
// coordinates in the U and V directions.
//     1   0  0  0
//     0   1  0  0
//     du dv  1  0 (du and dv change each frame)
//     0   0  0  1

D3DMATRIX M = D3DXMatrixIdentity(); // declared in d3dutil.h
M._31 = du;
M._32 = dv;
```

Creating texture coordinates as a linear function of a model's camera-space position

- Use the D3DTSS_TCI_CAMERASPACEPOSITION flag to instruct the system to pass the vertex position, in camera space, as input to a texture transformation.

```
// The input vertices have no texture coordinates, saving
// bandwidth. Three texture coordinates are generated by
// using vertex position in camera space (x, y, z).
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEPOSITION);
```

- Instruct the rasterizer to expect 2-D texture coordinates.

```
// Two output coordinates are used.
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTFF_COUNT2);
```

- Define and set a matrix that applies a linear function.

```
// Generate texture coordinates as linear functions
// so that:
//     u = Ux*x + Uy*x + Uz*x + Uw
//     v = Vx*x + Vy*x + Vz*x + Vw
// The matrix M for this case is:
//     Ux  Vx  0  0
//     Uy  Vy  0  0
//     Uz  Vz  0  0
//     Uw  Vw  0  0
```

```
SetTransform(D3DTS_TEXTURE0, &M);
```

Performing environment mapping with a cubic environment map

- Use the D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR flag to instruct the system to automatically generate texture coordinates as reflection vectors for cubic mapping.

```
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR);
```

- Instruct the rasterizer to expect texture coordinates with three elements.

```
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTFF_COUNT3);
```

Performing projective texturing

- Use the D3DTSS_TCI_CAMERASPACEPOSITION flag to instruct the system to pass the vertex position as input to a texture transformation matrix.

```
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEPOSITION);
```

- Create and apply the texture projection matrix. This is beyond the scope of this documentation, and is the topic of several industry articles.
- Instruct the rasterizer to expect three-element projected texture coordinates.

```
// Two output coordinates are used.
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTF_PROJECTED | D3DTTFF_COUNT3);
```

[Visual Basic]

The following list contains examples of ways you might use texture coordinate processing to achieve special texturing effects.

Animating textures (by translation or rotation) on a model

- Define 2-D texture coordinates in your vertex format.

```
' Use a single texture, with 2-D texture coordinates. This
' bit-pattern should be expanded to include position, normal,
' and color information as needed.
lFVFTex = (D3FVF_TEX1 Or D3DFVF_TEXCOORDSIZE2(0))
```

- Configure the rasterizer to use 2-D texture coordinates.

```
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2)
```

- Define and set an appropriate texture coordinate transformation matrix.

```
' M is a D3DMATRIX being set to translate texture
' coordinates in the U and V directions.
'    1   0  0  0
'    0   1  0  0
'    du dv  0  0 (du and dv change each frame)
'    0   0  0  0
dx.IdentityMatrix(M)
M.rc31 = du
M.ec32 = dv
```

Creating texture coordinates as a linear function of a model's camera-space position

- Use the D3DTSS_TCI_CAMERASPACEPOSITION flag to instruct the system to pass the vertex position, in camera space, as input to a texture transformation.

```
' The input vertices have NO texture coordinates, saving
' bandwidth. Three texture coordinates are generated by
' using vertex position in camera space (x, y, z).
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEPOSITION)
```

- Instruct the rasterizer to expect 2-D texture coordinates.

```
' Two output coordinates are used.
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2)
```

- Define and set a matrix that applies a linear function.

```
' Generate texture coordinates as linear functions
' so that:
'    u = Ux*x + Uy*x + Uz*x + Uw
'    v = Vx*x + Vy*x + Vz*x + Vw
' The matrix M for this case is:
'    Ux  Vx  0  0
'    Uy  Vy  0  0
'    Uz  Vz  0  0
'    Uw  Vw  0  0

SetTransform(D3DTS_TEXTURE0, M)
```

Performing environment mapping with a cubic environment map

- Use the D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR flag to instruct the system to automatically generate texture coordinates as reflection vectors for cubic mapping.

```
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR)
```

- Instruct the rasterizer to expect texture coordinates with three elements.

```
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT3)
```

Performing projective texturing

- Use the D3DTSS_TCI_CAMERASPACEPOSITION flag to instruct the system to pass the vertex position as input to a texture transformation matrix:

```
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEPOSITION)
```

- Create and apply the texture projection matrix. This is beyond the scope of this documentation, and is the topic of several industry articles.
- Instruct the rasterizer to expect three-element projected texture coordinates.

```
' Two output coordinates are used.
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTF_PROJECTED | D3DTTFF_COUNT3)
```

Texture Resources

This section presents information on creating and rendering with texture interface pointers. The information is organized in the following topics.

- Obtaining a Texture Surface Object
- Rendering with Texture Resources

Obtaining a Texture Surface Object

[C++]
Texture resources are implemented in the **IDirect3DTexture8** interface. To obtain a pointer to a texture interface, call the **IDirect3DDevice8::CreateTexture** method or any of the following D3DX functions.

- **D3DXCreateTexture**
- **D3DXCreateTextureFromFileA**
- **D3DXCreateTextureFromFileExA**
- **D3DXCreateTextureFromFileExW**
- **D3DXCreateTextureFromFileInMemory**
- **D3DXCreateTextureFromFileInMemoryEx**

- **D3DXCreateTextureFromFileW**
- **D3DXCreateTextureFromResourceA**
- **D3DXCreateTextureFromResourceExA**
- **D3DXCreateTextureFromResourceExW**
- **D3DXCreateTextureFromResourceW**

The following code example uses **D3DXCreateTextureFromFileA** to load a texture from Tiger.bmp.

```
// The following code example assumes that d3dDevice
// is a valid pointer to an IDirect3DDevice8 interface.

LPDIRECT3DTEXTURE8 pTexture;

D3DXCreateTextureFromFile( d3dDevice, "tiger.bmp", &pTexture);
```

The first parameter that **D3DXCreateTextureFromFile** accepts is a pointer to a **IDirect3DDevice8** interface. The second parameter tells Microsoft® Direct3D® the name of the file from which to load the texture. The third parameter takes the address of a pointer to a **IDirect3DTexture8** interface, representing the created texture object.

[Visual Basic]
Texture resources are implemented in the **Direct3DTexture8** class. To obtain a texture object, call the **Direct3DDevice8.CreateTexture** method or any of the following D3DX functions.

- **D3DX8.CreateTexture**
- **D3DX8.CreateTextureFromFile**
- **D3DX8.CreateTextureFromFileEx**
- **D3DX8.CreateTextureFromFileInMemory**
- **D3DX8.CreateTextureFromFileInMemoryEx**
- **D3DX8.CreateTextureFromResource**
- **D3DX8.CreateTextureFromResourceEx**

The following code example uses **D3DX.CreateTextureFromFile** to load a texture from Tiger.bmp.

```
'
' The following code example assumes that d3dDevice
' is a valid Direct3DDevice8 object and that g_D3DX is a valid
' D3DX8 object.
'
Dim Texture As Direct3DTexture8

Set Texture = g_D3DX.CreateTextureFromFile( d3dDevice, App.Path + "\tiger.bmp" )
```

The first parameter that **CreateTextureFromFile** accepts is a **Direct3DDevice8** object. The second parameter tells Microsoft® Direct3D® the name of the file from which to load the texture. This method returns a texture object, which is assigned to Texture.

Rendering with Texture Resources

Microsoft® Direct3D® supports multiple texture blending through the concept of texture stages. Each texture stage contains a texture and operations that can be performed on the texture. The textures in the texture stages form the set of current textures. For more information, see Multiple Texture Blending. The state of each texture is encapsulated in its texture stage.

[C++]
In a C++ application, the state of each texture must be set with the **IDirect3DDevice8::SetTextureStageState** method. Pass the stage number (0-7) as the value of the first parameter. Set the value of the second parameter to a member of the **D3DTEXTURESTAGESTATETYPE** enumerated type. The final parameter is the state value for the particular texture state.
Using texture interface pointers, your application can render a blend of up to eight textures. Set the current textures by invoking the **IDirect3DDevice8::SetTexture** method. Direct3D blends all current textures onto the primitives that it renders.

Note

The **SetTexture** method increments the reference count of the texture surface being assigned. When the texture is no longer needed, you should set the texture at the appropriate stage to NULL. If you fail to do this, the surface will not be released, resulting in a memory leak.

Your application can set the texture wrapping state for the current textures by calling the **IDirect3DDevice8::SetRenderState** method. Pass a value from D3DRS_WRAP0 through D3DRR_WRAP7 as the value of the first parameter, and use a combination of the D3DWRAPCOORD_0, D3DWRAPCOORD_1, D3DWRAPCOORD_2, and D3DWRAPCOORD_3 flags to enable wrapping in the u, v, or w directions.
Your application can also set the texture perspective and texture filtering states. See Texture Filtering and Texture Filtering State.

[Visual Basic]
In Microsoft® Visual Basic®, the state of each texture must be set with the **Direct3DDevice7.SetTextureStageState** method. Pass the stage number (0-7) as the value of the first parameter. Set the value of the second parameter to a member of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration. The final parameter is the state value for the particular texture state.
Applications can render a blend of up to eight textures. Set the current textures by invoking the **Direct3DDevice7.SetTexture** method. Direct3D blends all current textures onto the primitives that it renders.

Note

When the texture is no longer needed, you should set the texture at the appropriate stage to Nothing. If you fail to do this, the memory for the surface may be lost when your application closes.

Your application can set the texture wrapping state for the current textures by calling the **Direct3DDevice7.SetRenderState** method. Pass a value from D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7 as the value of the first parameter and use a combination of the D3DWRAPCOORD_0, D3DWRAPCOORD_1, D3DWRAPCOORD_2, and D3DWRAPCOORD_3 flags from the **CONST_D3D** enumeration to enable wrapping in the u or v directions. Your application can also set the texture perspective and texture filtering states. See Texture Filtering and Texture Filtering State.

Texture Filtering

When Microsoft® Direct3D® renders a primitive, it maps the 3-D primitive onto a 2-D screen. If the primitive has a texture, Direct3D must use that texture to produce a color for each pixel in the primitive's 2-D rendered image. For every pixel in the primitive's on-screen image, it must obtain a color value from the texture. This process is called texture filtering.

When a texture filter operation is performed, the texture being used is typically also being magnified or minified. In other words, it is being mapped onto a primitive image that is larger or smaller than itself. Magnification of a texture can result in many pixels being mapped to one texel. The result can be a chunky appearance. Minification of a texture often means that a single pixel is mapped to many texels. The resulting image can be blurry or aliased. To resolve these problems, some blending of the texel colors must be performed to arrive at a color for the pixel. Direct3D simplifies the complex process of texture filtering. It provides you with three types of texture filtering—linear filtering, anisotropic filtering, and mipmap filtering. If you select no texture filtering, Direct3D uses a technique called nearest-point sampling.

Each type of texture filtering has advantages and disadvantages. For instance, linear texture filtering can produce jagged edges or a chunky appearance in the final image. However, it is a computationally low-overhead method of texture filtering. Filtering with mipmaps usually produces the best results, especially when combined with anisotropic filtering. However, it requires the most memory of the techniques that Direct3D supports.

[C++]

Applications that use texture interface pointers should set the current texture filtering method by calling the **IDirect3Ddevice8::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass D3DTEXTUREMAGFILTER, D3DTEXTUREMINFILTER, or D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Pass a member

of the **D3DTEXTUREFILTERTYPE** enumerated type as the value in the third parameter to set the magnification, minification, or mipmapping filter.

[Visual Basic]
Microsoft® Visual Basic® applications should set the current texture filtering method by calling the **Direct3DDevice8.SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass CONST_D3DTEXTUREMAGFILTER, CONST_D3DTEXTUREMINFILTER, or CONST_D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Set the third parameter to the value from **CONST_D3DTEXTUREFILTERTYPE** to set the magnification, minification, or mipmapping filter.

This section presents the texture filtering methods that Direct3D supports. It is organized into the following topics.

- Nearest-Point Sampling
- Linear Texture Filtering
- Anisotropic Texture Filtering
- Texture Filtering With Mipmaps

[C++]

Note

Although the texture-filtering render states present in the **D3DRENDERSTATETYPE** enumerated type are superseded by texture stage states, the **IDirect3DDevice8::SetRenderState**, as opposed to the **IDirect3DDevice2** version, does not fail if you attempt to use them. Rather, the system maps the effects of these render states to the first stage in the multitexture cascade, stage 0. Applications should not mix the legacy render states with their corresponding texture stage states, as unpredictable results can occur.

Nearest-Point Sampling

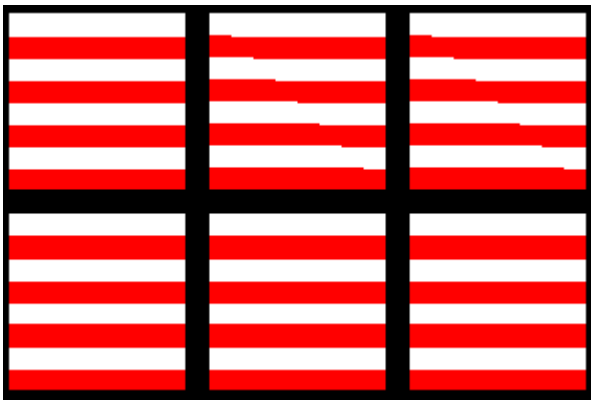
Applications are not required to use texture filtering. Microsoft® Direct3D® can be set so that it computes the texel address, which often does not evaluate to integers, and simply copies the color of the texel with the closest integer address. This process is called *nearest-point sampling*. This can be a fast and efficient way to process textures if the size of the texture is similar to the size of the primitive's image on the screen. If not, the texture must be magnified or minified. The result can be a chunky, aliased, or blurred image.

[C++]
Your C++ application can select nearest-point sampling by calling the **IDirect3DDevice8::SetTextureStageState** method. Set the value of the first

parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass D3DTEXTUREMAGFILTER, D3DTEXTUREMINFILTER, or D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Pass a member of the **D3DTEXTUREFILTERTYPE** enumerated type as the value in the third parameter to set the magnification, minification, or mipmapping filter.
For more information, see Texture Filtering State.

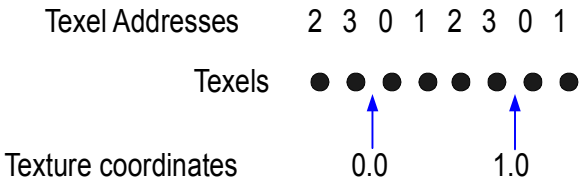
[Visual Basic]
Your Microsoft® Visual Basic® application can select nearest-point sampling by calling the **Direct3DDevice8.SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass CONST_D3DTEXTUREMAGFILTER, CONST_D3DTEXTUREMINFILTER, or CONST_D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Pass a member of the CONST_D3DTEXTUREFILTERTYPE enumeration type for the third parameter to set the magnification, minification, or mipmapping filter.
For more information, see Texture Filtering State.

You should use nearest-point sampling carefully, as it can sometimes cause graphic artifacts when the texture is sampled at the boundary between two texels. This boundary is the position along the texture (u or v) at which the sampled texel transitions from one texel to the next. When point sampling is used, the system chooses one sample texel or the other, and the result can change abruptly from one texel to the next texel as the boundary is crossed. This effect can appear as undesired graphic artifacts in the displayed texture. When linear filtering is used, the resulting texel is computed from both adjacent texels and smoothly blends between them as the texture index moves through the boundary.
This effect can be seen when mapping a very small texture onto a very large polygon: an operation often called *magnification*. For example, when using a texture that looks like a checkerboard, nearest-point sampling results in a larger checkerboard that shows distinct edges. By contrast, linear texture filtering results in an image where the checkerboard colors vary smoothly across the polygon.
In most cases, applications receive the best results by avoiding nearest-point sample wherever possible. The majority of hardware today is optimized for linear filtering, so your application should not suffer degraded performance. If the effect you desire absolutely requires the use of the nearest-point sampling—such as when using textures to display readable text characters—then your application should be extremely careful to avoid sampling at the texel boundaries, which might result in undesired effects. The following image shows what these artifacts can look like.



Notice that the two squares in the top-right of the group appear different than their neighbors, with diagonal offsets running through them. To avoid graphic artifacts like these, you must be familiar with Direct3D texture sampling rules for nearest-point filtering. Direct3D maps a floating-point texture coordinate ranging from [0.0, 1.0] (0.0 to 1.0, inclusive) to an integer texel space value ranging from [-0.5, *n* - 0.5], where *n* is the number of texels in a given dimension on the texture. The resulting texture index is rounded to the nearest integer. This mapping can introduce sampling inaccuracies at texel boundaries.

For a simple example, imagine an application that renders polygons with the D3DTADDRESS_WRAP texture addressing mode. Using the mapping employed by Direct3D, the *u* texture index maps as follows for a texture with a width of 4 texels.



Notice that the texture coordinates—0.0 and 1.0—for this illustration are exactly at the boundary between texels. Using the method by which Direct3D maps values, the texture coordinates range from [-0.5, 4 - 0.5], where 4 is the width of the texture. For this case, the sampled texel is the 0^{th} texel for a texture index of 1.0. However, if the texture coordinate was only slightly less than 1.0, the sampled texel would be the n^{th} texel instead of the 0^{th} texel.

The implication of this is that magnifying a small texture using texture coordinates of exactly 0.0 and 1.0 with nearest-point filtering on a screen-space aligned triangle results in pixels for which the texture map is sampled at the boundary between texels. Any inaccuracies in the computation of texture coordinates, however small, results in artifacts along the areas in the rendered image which correspond to the texel edges of the texture map.

Performing this mapping of floating point texture coordinates to integer texels with perfect accuracy is difficult, computationally expensive, and generally not necessary.

Most hardware implementations use an iterative approach for computing texture coordinates at each pixel location within a triangle. Iterative approaches tend to hide these inaccuracies because the errors are accumulated evenly during iteration.
The Direct3D reference rasterizer uses a direct-evaluation approach for computing texture indexes at each pixel location. Direct evaluation differs from the iterative approach in that any inaccuracy in the operation exhibits a more random error distribution. The result of this is that the sampling errors that occur at the boundaries can be more noticeable because the reference rasterizer does not perform this operation with perfect accuracy.
The best approach is to use nearest-point filtering only when necessary. When you must use it, it is recommended that you offset texture coordinates slightly from the boundary positions to avoid artifacts.

Linear Texture Filtering

Microsoft® Direct3D® uses a form of linear texture filtering called bilinear filtering. Like nearest-point sampling, bilinear texture filtering first computes a texel address, which is usually not an integer address. Bilinear filtering then finds the texel whose integer address is closest to the computed address. In addition, the Direct3D rendering module computes a weighted average of the texels that are immediately above, below, to the left of, and to the right of the nearest sample point.

[C++]
Select bilinear texture filtering by invoking the **IDirect3DDevice8::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass D3DTEXTUREMAGFILTER, D3DTEXTUREMINFILTER, or D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Set the third parameter to D3DTEXF_LINEAR. For more information, see Texture Filtering State.

[Visual Basic]
Select bilinear texture filtering by invoking the **Direct3DDevice8.SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass CONST_D3DTEXTUREMAGFILTER, CONST_D3DTEXTUREMINFILTER, or CONST_D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Set the third parameter to D3DTEXF_LINEAR. For more information, see Texture Filtering State.

Anisotropic Texture Filtering

Anisotropy is the distortion visible in the texels of a 3-D object whose surface is oriented at an angle with respect to the plane of the screen. When a pixel from an anisotropic primitive is mapped to texels, its shape is distorted. Microsoft® Direct3D®

measures the anisotropy of a pixel as the elongation—that is, length divided by width—of a screen pixel that is inverse-mapped into texture space.

[C++]
You can use anisotropic texture filtering in conjunction with linear texture filtering or mipmap texture filtering to improve rendering results. Your application enables anisotropic texture filtering by calling the **IDirect3DDevice8::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass D3DTEXTUREMAGFILTER, D3DTEXTUREMINFILTER, or D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Set the third parameter to D3DTEXF_ANISOTROPIC. For more information, see Texture Filtering State.
Your application must also set the degree of anisotropy to a value greater than one. Do this by calling the **IDirect3DDevice8::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are setting the degree of isotropy. Pass D3DTSS_MAXANISOTROPY as the value of the second parameter. The final parameter should be the degree of isotropy.
You can disable isotropic filtering by setting the degree of isotropy to one; any value larger than one enables it. Check the MaxAnisotropy flag in the **D3DCAPS8** structure to determine the possible range of values for the degree of anisotropy.

[Visual Basic]
You can use anisotropic texture filtering in conjunction with linear texture filtering or mipmap texture filtering to improve rendering results. Your application enables anisotropic texture filtering by calling the **Direct3DDevice8.SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass CONST_D3DTEXTUREMAGFILTER, CONST_D3DTEXTUREMINFILTER, or CONST_D3DTEXTUREMIPFILTER for the second parameter to set the magnification, minification, or mipmapping filter. Set the third parameter to D3DTEXF_ANISOTROPIC. For more information, see Texture Filtering State.
You must also set the degree of anisotropy to a value greater than one. Do this by calling the **Direct3DDevice8.SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are setting the degree of isotropy. Pass D3DTSS_MAXANISOTROPY as the value of the second parameter. The final parameter should be the degree of isotropy.
You can disable isotropic filtering by setting the degree of isotropy to one; any value larger than one enables it. Check the MaxAnisotropy flag in the **D3DCAPS8** type to determine the possible range of values for the degree of anisotropy.

Texture Filtering with Mipmaps

Mipmap textures are used in 3-D scenes to decrease the time required to render a scene. They also improve the scene's realism. However, they often require large amounts of memory.

This section presents the fundamentals of using mipmap textures in 3-D scenes. Information is organized in the following topics.

- What Is a Mipmap?
- Creating a Set of Mipmaps
- Selecting and Displaying a Mipmap

What Is a Mipmap?

A mipmap is a sequence of textures, each of which is a progressively lower resolution representation of the same image. The height and width of each image, or level, in the mipmap is a power of two smaller than the previous level. Mipmaps do not have to be square.

A high-resolution mipmap image is used for objects that are close to the user. Lower-resolution images are used as the object appears farther away. Mipmapping improves the quality of rendered textures at the expense of using more memory.

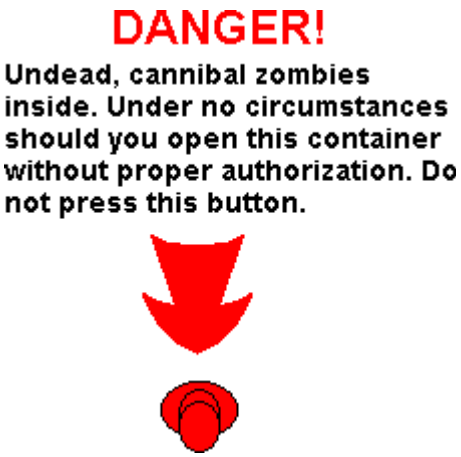
Microsoft® Direct3D® represents mipmaps as a chain of attached surfaces. The highest resolution texture is at the head of the chain and has the next level of the mipmap as an attachment. In turn, that level has an attachment that is the next level in the mipmap, and so on, down to the lowest resolution level of the mipmap.

The following illustrations shows an example of these levels. The bitmap textures represent a sign on a container in a 3-D, first-person game. When created as a mipmap, the highest-resolution texture is first in the set. Each succeeding texture in the mipmap set is smaller in height and width by a power of 2. In this case, the maximum-resolution mipmap is 256 pixels by 256 pixels. The next, texture is 128x128. The last texture in the chain is 64x64.

This sign has a maximum distance from which it is visible. If the user begins far away from the sign, the game displays the smallest texture in the mipmap chain, which in this case the 64x64 texture.

As the user moves the point of view closer to the sign, progressively higher-resolution textures in the mipmap chain are used. The resolution in the following illustration is 128x128.

The highest-resolution texture is used when the user's point of view is at the minimum allowable distance from the sign.



This is a computationally lower-overhead way of simulating perspective for textures. Rather than render a single texture at many resolutions, it is faster to use multiple textures at varying resolutions.
Direct3D can assess which texture in a mipmap set is the closest resolution to the desired output, and it can map pixels into its texel space. If the resolution of the final image is between the resolutions of the textures in the mipmap set, Direct3D can examine texels in both mipmaps and blend their color values together.
To use mipmaps, your application must build a set of mipmaps. For details, see Creating a Set of Mipmaps. Applications apply mipmaps by selecting the mipmap set as the first texture in the set of current textures. For more information, see Multiple Texture Blending.
Next, your application must set the filtering method that Direct3D uses to sample texels. The fastest method of mipmap filtering is to have Direct3D select the nearest texel. Use the D3DTFP_POINT enumerated value to select this. Direct3D can produce better filtering results if your application uses the D3DTFP_LINEAR enumerated value. This selects the nearest mipmap, and then computes a weighted average of the texels surrounding the location in the texture to which the current pixel maps.

Creating a Set of Mipmaps

[C++]
The following example shows how your application can call the **IDirect3DDevice8::CreateTexture** method to build a chain of five mipmap levels: 256×256, 128×128, 64×64, 32×32, and 16×16.

```
// This code example assumes that the variable d3dDevice is a
// valid pointer to a IDirect3DDevice8 interface.

IDirect3DTexture8 * pMipMap;
```

```
d3dDevice->CreateTexture(256, 256, 5, 0, D3DFMT_R8G8B8, D3DPOOL_MANAGED,
&pMipMap);
```

The first two parameters that are accepted by **CreateTexture** are the size and width of the top-level texture. The third parameter specifies the number of levels in the texture. If you set this to zero, Microsoft® Direct3D® creates a chain of surfaces, each a power of two smaller than the previous one, down to the smallest possible size of 1x1. The fourth parameter specifies the usage for this resource; in this case, 0 is specified to indicate no specific usage for the resource. The fifth parameter specifies the surface format for the texture. Use a value from the **D3DFORMAT** enumerated type for this parameter. The sixth parameter specifies a member of the **D3DPOOL** enumerated type indicating the memory class into which to place the created resource. Unless you are using dynamic textures, D3DPOOL_MANAGED is recommended. The final parameter takes the address of a pointer to a **IDirect3DTexture8** interface.

[Visual Basic]
The following example shows how your application can call **Direct3DDevice8.CreateTexture** to build a chain of five mipmap levels: 256×256, 128×128, 64×64, 32×32, and 16×16.

```
' This code example assumes that the variable m_D3D contains
' a valid reference to a Direct3DDevice8 object.
Dim DDMipmap as Direct3DTexture8

Set DDMipmap = m_D3D.CreateTexture(256, 256, 5, 0, D3DFMT_R8G8B8,
D3DPOOL_DEFAULT)
```

The first two parameters that are accepted by **CreateTexture** are the size and width of the top-level texture. The third parameter specifies the number of levels in the texture. If you set this to zero, Microsoft® Direct3D® creates a chain of surfaces, each a power of two smaller than the previous one, down to the smallest possible size of 1x1. The fourth parameter specifies the usage for this resource; in this case, 0 is specified to indicate no specific usage for the resource. The fifth parameter specifies the surface format for the texture. Use a value from the **CONST_D3DFORMAT** enumerated type for this parameter. The sixth parameter specifies a member of the **CONST_D3DPOOL** enumerated type indicating the memory class into which to place the created resource. Unless you are using dynamic textures, D3DPOOL_MANAGED is recommended.

Note

Each surface in a mipmap chain has dimensions that are one-half that of the previous surface in the chain. If the top-level mipmap has dimensions of 256×128, the dimensions of the second-level mipmap are 128×64, the third-level is 64×32, and so on, down to 1×1. You cannot request a number of mipmap levels in **Levels** that would cause either the width or height of any mipmap in the chain to be smaller than 1. In the simple case of a 4×2 top-level

mipmap surface, the maximum value allowed for **Levels** is three. The top-level dimensions are 4×2, the second-level dimensions are 2×1, and the dimensions for the third level are 1×1. A value larger than 3 in **Levels** results in a fractional value in the height of the second-level mipmap, and is therefore disallowed.

Selecting and Displaying a Mipmap

[C++]
Call the **IDirect3DDevice8::SetTexture** method to set the mipmap texture set as the first texture in the list of current textures. For more information, see Multiple Texture Blending.
After your application selects the mipmap texture set, it must assign values from the **D3DTEXTUREFILTERTYPE** enumerated type to the D3DTSS_MIPFILTER texture stage state. Microsoft® Direct3D® then automatically performs mipmap texture filtering. Enabling mipmap texture filtering is demonstrated in the following code example.

```
d3dDevice->SetTexture(0, pMipMap);
d3dDevice->SetTextureStageState(0, D3DTSS_MIPFILTER, D3DTEXF_POINT);
```

Your application can also manually traverse a chain of mipmap surfaces by using the **IDirect3DTexture8::GetSurfaceLevel** method and specifying the mipmap level to retrieve. The following example traverses a mipmap chain from highest to lowest resolutions.

```
IDirect3DSurface8 * pSurfaceLevel;

for (int iLevel = 0; iLevel < pMipMap->GetLevelCount(); iLevel++)
{
    pMipMap->GetSurfaceLevel(iLevel, &pSurfaceLevel);

    //Process this level.

    pSurfaceLevel->Release();
}
```

Applications need to manually traverse a mipmap chain to load bitmap data into each surface in the chain. This is typically the only reason to traverse the chain. An application can retrieve the number of levels in a mipmap by calling **IDirect3DBaseTexture8::GetLevelCount**.

[Visual Basic]
Call the **Direct3DDevice8.SetTexture** method to set the mipmap texture set as the first texture in the list of current textures. For more information, see Multiple Texture Blending.
After your application selects the mipmap texture set, it must assign values from the **CONST_D3DTEXTUREFILTERTYPE** enumerated type to the D3DTSS_MIPFILTER texture stage state. Microsoft® Direct3D® then automatically

performs mipmap texture filtering. Enabling mipmap texture filtering is demonstrated in the following code example.

```
m_D3D.SetTexture(0, DDMipMap)
m_D3D.SetTextureStateState(0, D3DTSS_MIPFILTER, D3DTEXF_POINT)
```

Your application can also manually traverse a chain of mipmap surfaces by using the **Direct3DTexture8.GetSurfaceLevel** method and specifying the mipmap level to retrieve. The following example traverses a mipmap chain from highest to lowest resolutions.

```
Dim DDLevel As Direct3DSurface8
Dim cLevels As Long

cLevels = DDMipMap.GetLevelCount

For iLevel = 0 to cLevels
   Set DDLevel = DDMipMap.GetSurfaceLevel(iLevel)

   'Process this level.

   Set DDLevel = Nothing
Next
```

Applications must manually traverse a mipmap chain to load bitmap data into each surface in the chain. This is typically the only reason to traverse the chain. An application can retrieve the number of levels in a mipmap texture by calling **Direct3DBaseTexture8.GetLevelCount**.

Texture Wrapping

This section presents information on wrapping textures around 3-D primitives. The discussion is divided into the following topics.

- What Is Texture Wrapping?
- Using Texture Wrapping

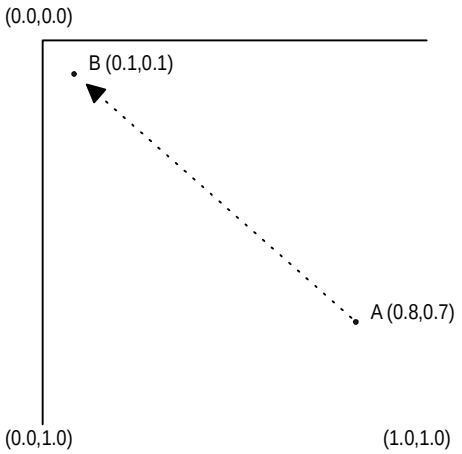
Note

Do not confuse texture wrapping with the similarly named texture addressing modes. For more information, see Texture Addressing Modes and Texture Wrapping and Texture Addressing Modes.

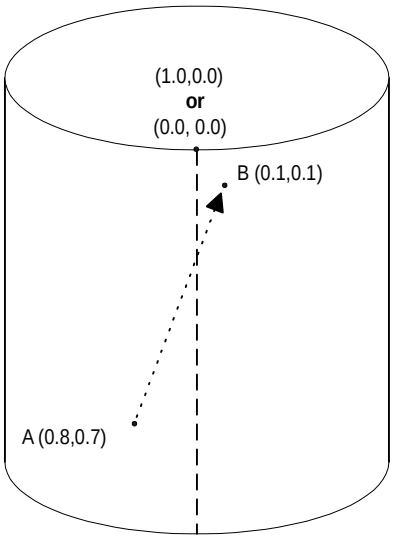
What Is Texture Wrapping?

In short, texture wrapping changes the basic way that Microsoft® Direct3D® rasterizes textured polygons using the texture coordinates specified for each vertex. While rasterizing a polygon, the system interpolates between the texture coordinates at each

of the polygon's vertices to determine the texels that should be used for every pixel of the polygon. Normally, the system treats the texture as a 2-D plane, interpolating new texels by taking the shortest route from point A within a texture to point B. If point A represents the u, v position (0.8, 0.1), and point B is at (0.1,0.1), the line of interpolation looks like the following illustration.

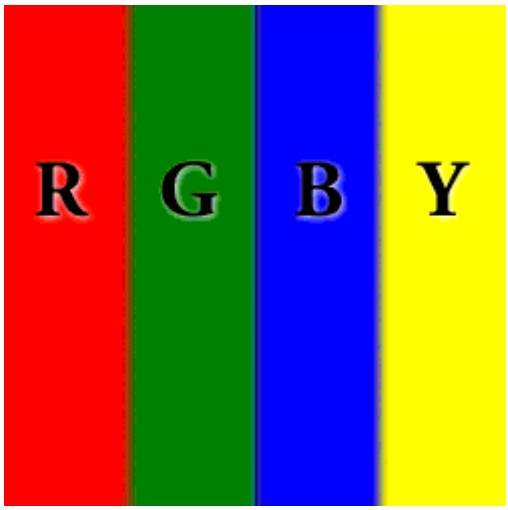


Note that the shortest distance between A and B in this illustration runs roughly through the middle of the texture. Enabling u- or v-texture coordinate wrapping changes how Direct3D perceives the shortest route between texture coordinates in the u- and v-directions. By definition, texture wrapping causes the rasterizer to take the shortest route between texture coordinate sets, assuming that 0.0 and 1.0 are coincident. The last bit is the tricky part: you can imagine that enabling texture wrapping in one direction causes the system to treat a texture as though it were wrapped around a cylinder. For example, consider the following illustration.

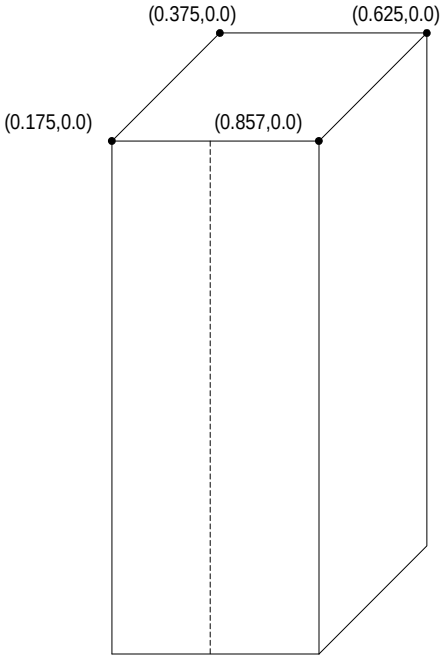


This diagram shows how wrapping in the u- direction affects how the system interpolates texture coordinates. Using the same points as in the example for normal, or nonwrapped, textures, you can see that the shortest route between points A and B is no longer across the middle of the texture; it's now across the border where 0.0 and 1.0 exist together. Wrapping in the v-direction is similar, except that it wraps the texture around a cylinder that is lying on its side. Wrapping in both the u- and v-directions is a more complex. In this situation, you can envision the texture as a torus, or doughnut.

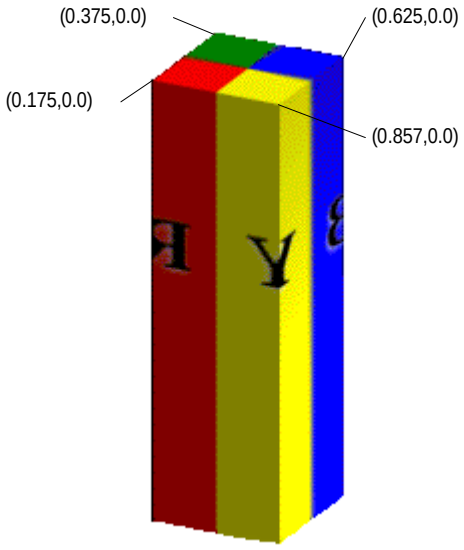
The most common practical application for texture wrapping is to perform environment mapping. Usually, an object textured with an environment map appears very reflective, showing a mirrored image of the object's surroundings in the scene. For the sake of this discussion, picture a room with four walls, each one painted with a letter R, G, B, Y and the corresponding colors: red, green, blue, and yellow. The environment map for such a simple room might look like the following illustration.



Imagine that the room's ceiling is held up by a perfectly reflective, four-sided, pillar. Mapping the environment map texture to the pillar is simple; making the pillar look as though it's reflecting the letters and colors is not as easy. The following diagram shows a wire frame of the pillar with the applicable texture coordinates listed near the top vertices. The seam where wrapping will cross the edges of the texture is shown with a dotted line.



With wrapping enabled in the u- direction, the textured pillar shows the colors and symbols from the environment map appropriately and, at the seam in the front of the texture, the rasterizer properly chooses the shortest route between the texture coordinates, assuming that u-coordinates 0.0 and 1.0 share the same location. The textured pillar looks like the following illustration.



If texture wrapping isn't enabled, the rasterizer does not interpolate in the direction needed to generate a believable, reflected image. Rather, the area at the front of the pillar contains a horizontally compressed version of the texels between u- coordinates 0.175 and 0.875, as they pass through the center of the texture. The wrap effect is ruined.

Using Texture Wrapping

[C++]
To enable texture wrapping, call the **IDirect3DDevice8::SetRenderState** method as shown in the code example below.

```
d3dDevice->SetRenderState(D3DRS_WRAP0, D3DWRAPCOORD_0);
```

The first parameter accepted by **SetRenderState** is a render state to set. Specify one of the D3DRS_WRAP0 through D3DRS_WRAP7 enumerated values which specify which texture level to set the wrapping for. Specify the D3DWRAPCOORD_0 through D3DWRAPCOORD_3 flags in the second parameter to enable texture wrapping in the corresponding direction, or combine them to enable wrapping in multiple directions. If you omit a flag, texture wrapping in the corresponding

direction is disabled. To disable texture wrapping for a set of texture coordinates, set the value for the corresponding render state to 0.

[Visual Basic]
To enable texture wrapping, call the **Direct3DDevice8.SetRenderState** method as shown in the code example below.

```
m_D3D.SetRenderState(D3DRS_WRAP0, D3DWRAPCOORD_0)
```

The first parameter accepted by **SetRenderState** is a render state to set. Specify one of the D3DRS_WRAP0 through D3DRS_WRAP7 enumerated values which specify which texture level to set the wrapping for. Specify the D3DWRAPCOORD_0 through D3DWRAPCOORD_3 flags in the second parameter to enable texture wrapping in the corresponding direction, or combine them to enable wrapping in multiple directions. If you omit a flag, texture wrapping in the corresponding direction is disabled. To disable texture wrapping for a set of texture coordinates, set the value for the corresponding render state to 0.

Texture Blending

Microsoft® Direct3D® can produce transparency effects by blending a texture with a primitive's color. It can also blend multiple textures onto a primitive. This section presents information on how texture blending is done. It is divided into the following topics.

- Alpha Texture Blending
- Multipass Texture Blending
- Multiple Texture Blending
- Light Mapping With Textures

[C++]
To use texture blending, your application should first check if the user's hardware supports it. This information is found in the **TextureCaps** member of the **D3DCAPS8** structure. For details on how to query the user's hardware for texture blending capabilities, see **IDirect3DDevice8::GetDeviceCaps**.

[Visual Basic]
To use texture blending, your Microsoft® Visual Basic® application should first check if the user's hardware supports it. This information is found in the **TextureCaps** member of the **D3DCAPS8** type. For details on how to query the user's hardware for texture blending capabilities, see **Direct3DDevice8.GetDeviceCaps**.

Alpha Texture Blending

When Microsoft® Direct3D® renders a primitive, it generates a color for the primitive based on the primitive's material, or the colors of its vertices, and lighting information. For details, see Introduction to Lighting and Materials. If an application enables texture blending, Direct3D must then combine the color value of the processed polygon pixel with the pixel already stored in the frame buffer. Direct3D uses the following formula to determine the final color for each pixel in the primitive's image.

FinalColor = *TexelColor* × *SourceBlendFactor* + *PixelColor* × *DestBlendFactor*

In this formula, *FinalColor* is the final pixel color that is output to the target rendering surface. *TexelColor* represents the incoming color value, after texture filtering, that corresponds to the current pixel. For details on how Direct3D maps texels to pixels, see Texture Filtering. *SourceBlendFactor* is a calculated value that Direct3D uses to determine the percentage of the incoming color value to apply to the final color. *PixelColor* is the color of the pixel currently stored in the primitive's image. *DestBlendFactor* represents the percentage of the current pixel's color that will be used in the final rendered pixel. The values of *SourceBlendFactor* and *DestBlendFactor* range from 0.0 to 1.0 inclusive.

As you can see from the preceding formula, the final rendered pixel is not rendered as transparent if the *SourceBlendFactor* is D3DBLEND_ONE and the *DestBlendFactor* is D3DBLEND_ZERO. It is completely transparent if the *SourceBlendFactor* is D3DBLEND_ZERO and the *DestBlendFactor* is D3DBLEND_ONE. If an application sets these factors to any other values, the resulting final rendered pixel is blended with some degree of transparency.

After texture filtering, every pixel color value has red, green, and blue color values. By default, Direct3D uses D3DBLEND_SRCALPHA as the *SourceBlendFactor* and D3DBLEND_INVSRCALPHA as the *DestBlendFactor*. Therefore, applications can control the transparency of processed pixels by setting the alpha values in textures.

[C++]

A C++ application controls the blending factors with the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states. Invoke the **IDirect3DDevice8::SetRenderState** method and pass either render state value as the value of the first parameter. The second parameter must be a member of the **D3DBLEND** enumerated type.

[Visual Basic]

A Visual Basic application controls the blending factors with the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states. Invoke the **Direct3DDevice8.SetRenderState** method and pass either render state value as the value of the first parameter. The second parameter must be a member of the **CONST_D3DBLEND** enumeration.

Multipass Texture Blending

Microsoft® Direct3D® applications can achieve numerous special effects by applying various textures to a primitive over the course of multiple rendering passes. The

common term for this is multipass texture blending. A typical use for multipass texture blending is to emulate the effects of complex lighting and shading models by applying multiple colors from several different textures. One such application is called light mapping. For more information, see Light Mapping With Textures.

Note

Some devices are capable of applying multiple textures to primitives in a single pass. For details, see Multiple Texture Blending.

If the user's hardware does not support multiple texture blending, your application can use multipass texture blending to achieve the same visual effects. However, the application cannot sustain the frame rates that are possible when using multiple texture blending.

[C++]

To perform multipass texture blending in a C++ application

1. Set a texture in texture stage 0 by calling the **IDirect3DDevice8::SetTexture** method.
2. Select the desired color and alpha blending arguments and operations with the **IDirect3DDevice8::SetTextureStageState** method. The default settings are well-suited for multipass texture blending.
3. Render the appropriate objects in the scene.
4. Set the next texture in texture stage 0.
5. Set the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states to adjust the source and destination blending factors as needed. The system blends the new textures with the existing pixels in the render-target surface according to these parameters.
6. Repeat Steps 3, 4, and 5 with as many textures as needed.

[Visual Basic]

To perform multipass texture blending in a Visual Basic application

1. Set a texture in texture stage 0 by calling the **Direct3DDevice8.SetTexture** method.
2. Select the desired color and alpha blending arguments and operations with the **Direct3DDevice8.SetTextureStageState** method. The default settings are well-suited for multipass texture blending.
3. Render the appropriate objects in the scene.
4. Set the next texture in texture stage 0.
5. Set the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states to adjust the source and destination blending factors as needed. The system blends the new textures with the existing pixels in the render-target surface according to these parameters.

6. Repeat Steps 3, 4, and 5 with as many textures as needed.

Multiple Texture Blending

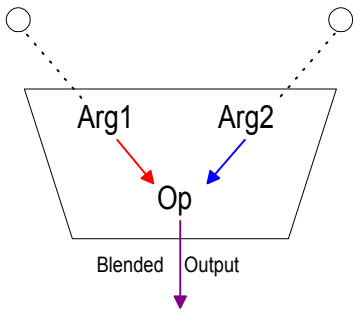
Microsoft® Direct3D® can blend as many as eight textures onto primitives in a single pass. The use of multiple texture blending can profoundly increase the frame rate of a Direct3D application. An application employs multiple texture blending to apply textures, shadows, specular lighting, diffuse lighting, and other special effects in a single pass.

To blend multiple textures, an application assigns textures to the set of current textures, and then creates blending stages. The following topics contain information on how these steps are accomplished.

- Texture Stages and the Texture Blending Cascade
- Texture Blending Operations and Arguments
- Assigning the Current Textures
- Creating Blending Stages

Texture Stages and the Texture Blending Cascade

Microsoft® Direct3D® supports single-pass multiple texture blending through the use of *texture stages*. A texture stage takes two arguments and performs a blending operation on them, passing on the result for further processing or for rasterization. You can visualize a texture stage as shown in the following illustration.

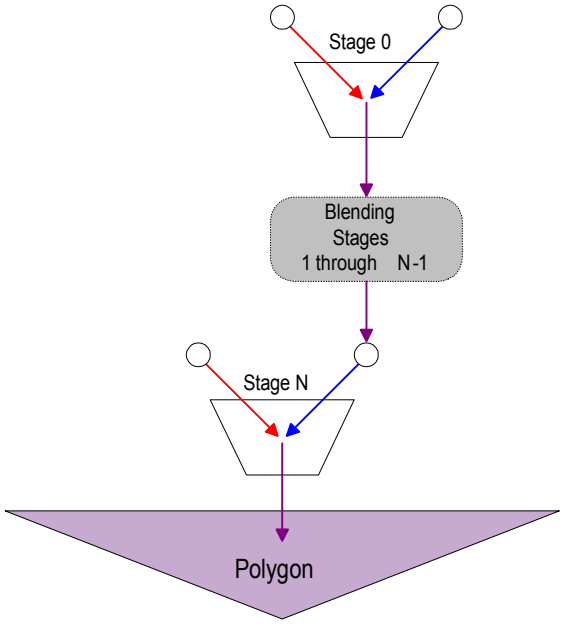


As the preceding illustration shows, texture stages blend two arguments by using a specified operator. Common operations include simple modulation or addition of the color or alpha components of the arguments, but more than two dozen operations are supported. The arguments for a stage can be an associated texture, the iterated color or alpha (iterated during Gouraud shading), arbitrary color and alpha, or the result from the previous texture stage. For more information, see Texture Blending Operations and Arguments.

Note

Direct3D distinguishes color blending from alpha blending. Applications set blending operations and arguments for color and alpha individually, and the results of those settings are independent of one another.

The combination of arguments and operations used by multiple blending stages define a simple flow-based blending language. The results from one stage flow down to another stage, from that stage to the next, and so on. The concept of results flowing from stage to stage to eventually be rasterized on a polygon is often called the *texture blending cascade*. The following illustration shows how individual texture stages make up the texture blending cascade.



Each stage in a device has a zero-based index. Direct3D allows up to eight blending stages, although you should always check device capabilities to determine how many stages the current hardware supports. The first blending stage is at index 0, the second is at 1, and so on, up to index 7. The system blends stages in increasing index order. Use only the number of stages you need; the unused blending stages are disabled by default. So, if your application only uses the first two stages, it need only set operations and arguments for stage 0 and 1. The system blends the two stages, and ignores the disabled stages.

Optimization Note

If your application varies the number of stages it uses for different situations—such as four stages for some objects, and only two for others—you don't need to explicitly disable all previously used stages. If you disable the color operation for the first unused stage, all stages with a higher index will not be applied. You can disable texture mapping altogether by setting the color operation for the first texture stage (stage 0).

Texture Blending Operations and Arguments

Applications associate a blending stage with each texture in the set of current textures. As mentioned in Texture Stages and the Texture Blending Cascade, Microsoft® Direct3D® evaluates each blending stage in order, beginning with the first texture in the set and ending with the eighth.

[C++]
Direct3D applies the information from each texture in the set of current textures to the blending stage that is associated with it. Applications control what information from a texture stage is used by calling **IDirect3DDevice8::SetTextureStageState**. You can set separate operations for the color and alpha channels, and each operation uses two arguments. Specify color channel operations by using the D3DTSS_COLOROP stage state; specify alpha operations by using D3DTSS_ALPHAOP. Both stage states use values from the **D3DTEXTUREOP** enumerated type.
Texture blending arguments use the D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHARG1, and D3DTSS_ALPHARG2 members of the **D3DTEXTURESTAGESTATETYPE** enumerated type. The corresponding argument values are identified using texture argument flags.

Note

You can disable a texture stage—and any subsequent texture blending stages in the cascade—by setting the color operation for that stage to D3DTOP_DISABLE. Disabling the color operation effectively disables the alpha operation as well. Alpha operations cannot be disabled when color operations are enabled. Setting the alpha operation to D3DTOP_DISABLE when color blending is enabled causes undefined behavior.

To determine the supported texture blending operations of a device, query the **TextureCaps** member of the **D3DCAPS8** structure.

[Visual Basic]
Direct3D applies the information from each texture in the set of current textures to the blending stage that is associated with it. Applications control what information from a texture stage is used by calling **Direct3DDevice8.SetTextureStageState**. You can set separate operations for the color and alpha channels, and each operation uses two arguments. Specify color channel operations by using the D3DTSS_COLOROP stage state, and alpha operations by using D3DTSS_ALPHAOP. Both stage states use values from the **CONST_D3DTEXTUREOP** enumeration.
Texture blending arguments use the D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHARG1, and D3DTSS_ALPHARG2 members of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration. The corresponding argument values are identified using texture argument flags.

Note

You can disable a texture stage—and any subsequent texture blending stages in the cascade—by setting the color operation for that stage to D3DTOP_DISABLE. Disabling the color operation effectively disables the alpha

operation as well. Alpha operations cannot be disabled when color operations are enabled. Setting the alpha operation to D3DTOP_DISABLE when color blending is enabled causes undefined behavior.

To determine the supported texture blending operations of a device, query the **TextureCaps** member of the D3DCAPS8 type.

Assigning the Current Textures

Microsoft® Direct3D® maintains a list of up to eight current textures. It blends these textures onto all the primitive it renders. Only textures created as texture interface pointers can be used in the set of current textures.

[C++]
Applications call the **IDirect3DDevice8::SetTexture** method to assign textures into the set of current textures. The first parameter must be from the a number in the range of 0-7, inclusive. Pass the texture interface pointer as the second parameter.
The following C++ code example demonstrates how a texture can be assigned to the set of current textures.

```
// This code example assumes that the variable lpd3dDev is a
// valid pointer to an IDirect3DDevice8 interface and pTexture
// is a valid pointer to an IDirect3DBaseTexture8 interface.

// Set the third texture.
d3dDevice->SetTexture(2, pTexture);
```

[Visual Basic]
Applications written in Microsoft® Visual Basic® call **Direct3DDevice8.SetTexture** to assign textures to the set of current textures. The first parameter must be from the a number in the range of 0-7, inclusive. Pass the texture interface pointer as the second parameter.
The following Visual Basic code example demonstrates how a texture can be assigned into the set of current textures.

```
' This code example assumes that the variable d3dDevice is a
' valid reference to a Direct3DDevice8 object and d3dTexture
' is a valid reference to a Direct3DBaseTexture8 object.

' Set the third texture.
Call d3dDevice.SetTexture(2, d3dTexture)
```

Note

Software devices do not support assigning a texture to more than one texture stage at a time.

Creating Blending Stages

[C++]
A blending stage is a set of texture operations and their arguments that define how textures are blended. When making a blending stage, C++ applications invoke the **IDirect3DDevice8::SetTextureStageState** method. The first call specifies the operation that is performed. Two additional invocations define the arguments to which the operation is applied. The following code example illustrates the creation of a blending stage.

```
// This example assumes that lpD3DDev is a valid pointer to an
// IDirect3DDevice8 interface.

// Set the operation for the first texture.
d3dDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_ADD);

// Set argument 1 to the texture color.
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);

// Set argument 2 to the iterated diffuse color.
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
```

Texel data in textures contain color and alpha values. Applications can define separate operations for both color and alpha values in a single blending stage. Each operation, color and alpha, has its own arguments. For details, see **D3DTEXTURESTAGESTATETYPE**.
Although not part of the Microsoft® Direct3D® API, you can insert the following macros into your application to abbreviate the code required for creating texture blending stages.

```
#define SetTextureColorStage( dev, i, arg1, op, arg2 )    \
  dev->SetTextureStageState( i, D3DTSS_COLOROP, op);     \
  dev->SetTextureStageState( i, D3DTSS_COLORARG1, arg1 ); \
  dev->SetTextureStageState( i, D3DTSS_COLORARG2, arg2 );

#define SetTextureAlphaStage( dev, i, arg1, op, arg2 )    \
  dev->SetTextureStageState( i, D3DTSS_ALPHAOP, op);     \
  dev->SetTextureStageState( i, D3DTSS_ALPHARG1, arg1 );  \
  dev->SetTextureStageState( i  D3DTSS_ALPHARG2, arg2 );
```

[Visual Basic]
A blending stage is a set of texture operations and their arguments that define how textures are blended. When making a blending stage, Microsoft® Visual Basic® applications invoke the **Direct3DDevice8.SetTextureStageState** method. The first call specifies the operation that is performed. Two additional invocations define the arguments to which the operation is applied. The following code example illustrates the creation of a blending stage.

```
' This example assumes that d3dDevice is a valid reference to a
' Direct3DDevice7 object.

' Set the color operation for the first texture.
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_ADD)

' Set argument 1 to the texture color.
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE)

' Set argument 2 to the iterated diffuse color.
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE)
```

Texel data in textures contains color and alpha values. Applications can define separate operations for both color and alpha values in a single blending stage. Each operation, color and alpha, has its own arguments. For details, see **CONST_D3DTEXTURESTAGESTATETYPE**.

Light Mapping with Textures

For an application to realistically render a 3-D scene, it must take into account the effect that light sources have on the appearance of the scene. Although techniques such as flat and Gouraud shading are valuable tools in this respect, they can be insufficient for your needs. Microsoft® Direct3D® supports multipass and multiple texture blending. These capabilities enable your application to render scenes with a more realistic appearance than scenes rendered with shading techniques alone. By applying one or more light maps, your application can map areas of light and shadow onto its primitives.

A light map is a texture or group of textures that contains information about lighting in a 3-D scene. You can store the lighting information in the alpha values of the light map, in the color values, or in both.

If you implement light mapping using multipass texture blending, your application should render the light map onto its primitives on the first pass. It should use a second pass to render the base texture. The exception to this is specular light mapping. In that case, render the base texture first; then add the light map.

Multiple texture blending enables your application to render the light map and the base texture in one pass. If the user's hardware provides for multiple texture blending, your application should take advantage of it when performing light mapping. This significantly improves your application's performance.

Using light maps, a Direct3D application can achieve a variety of lighting effects when it renders primitives. It can map not only monochrome and colored lights in a scene, but it can also add details such as specular highlights and diffuse lighting. Information on using Direct3D texture blending to perform light mapping is presented in the following topics.

- Monochrome Light Maps
- Color Light Maps

- Specular Light Maps
- Diffuse Light Maps

Monochrome Light Maps

Some older 3-D accelerator boards do not support texture blending using the alpha value of the destination pixel. See Alpha Texture Blending for more information. These adapters also generally do not support multiple texture blending. If your application is running on an adapter such as this, it can use multipass texture blending to perform monochrome light mapping.
To perform monochrome light mapping, an application stores the lighting information in the alpha data of its light map textures. The application uses the texture filtering capabilities of Microsoft® Direct3D® to perform a mapping from each pixel in the primitive's image to a corresponding texel in the light map. It sets the source blending factor to the alpha value of the corresponding texel.

[C++]
The following C++ code example illustrates how an application can use a texture as a monochrome light map.

```
// This example assumes that d3dDevice is a valid pointer to an
// IDirect3DDevice7 interface and that lptexLightMap is a valid
// pointer to a texture that contains monochrome light map data.

// Set the light map texture as the current texture.
d3dDevice->SetTexture(0, lptexLightMap);

// Set the color operation.
d3dDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);

// Set argument 1 to the color operation.
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG1,
    D3DTA_TEXTURE | D3DTA_ALPHAREPLICATE);
```

[Visual Basic]
The following Microsoft® Visual Basic® code example illustrates how an application can use a texture as a monochrome light map.

```
' This example assumes that D3DDevice is a valid reference to a
' Direct3DDevice8 object and that texLightMap is a valid reference
' to a Direct3DTexture8 object that contains monochrome light map data.

' Set the light map texture as the current texture.
Call D3DDevice.SetTexture(0, texLightMap)

' Set the color operation.
Call D3DDevice.SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG1)
```

```
' Set argument 1 to the color operation.
Call D3DDevice.SetTextureStageState(0, D3DTSS_COLORARG1, _
 D3DTA_TEXTURE Or D3DTA_ALPHAREPLICATE)
```

Because display adapters that do not support destination alpha blending usually do not support multiple texture blending, this example sets the light map as the first texture, which is available on all 3-D accelerator cards. The sample code sets the color operation for the texture's blending stage to blend the texture data with the primitive's existing color. It then selects the first texture and the primitive's existing color as the input data.

Color Light Maps

Your application will usually render 3-D scenes more realistically if it uses colored light maps. A colored light map uses the RGB data in the light map for its lighting information.

[C++]

The following C++ code example demonstrates light mapping with RGB color data.

```
// This example assumes that d3dDevice is a valid pointer to an
// IDirect3DDevice8 interface and that lptexLightMap is a valid
// pointer to a texture that contains RGB light map data.

// Set the light map texture as the first texture.
d3dDevice->SetTexture(0, lptexLightMap);

d3dDevice->SetTextureStageState( 0,D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState( 0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );
```

This example sets the light map as the first texture. It then sets the state of the first blending stage to modulate the incoming texture data. It uses the first texture and the current color of the primitive as the arguments to the modulate operation.

[Visual Basic]

The following Microsoft® Visual Basic® code example demonstrates light mapping with RGB color data.

```
' This example assumes that D3DDevice is a valid reference to a
' Direct3DDevice8 object and that texLightMap is a valid reference
' to a 3DBaseTexture8 object that contains RGB light map data.

' Set the light map texture as the first texture.
D3DDevice.SetTexture 0, texLightMap
```

```
D3DDevice.SetTextureStageState 0, D3DTSS_COLOROP, D3DTOP_MODULATE
D3DDevice.SetTextureStageState 0, D3DTSS_COLORARG1, D3DTA_TEXTURE
D3DDevice.SetTextureStageState 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE
```

This sample sets the light map as the first texture. It then sets the state of the first blending stage to modulate the incoming texture data. It uses the first texture and the current color of the primitive as the arguments to the modulate operation.

Specular Light Maps

When illuminated by a light source, shiny objects—those that use highly reflective materials—receive specular highlights. In some cases, the specular highlights produced by the lighting module is not accurate. To produce a more appealing highlight, many Microsoft® Direct3D® applications apply specular light maps to primitives.

To perform specular light mapping, first modulate the specular light map with the primitive's existing texture. Then add the monochrome or RGB light map.

[C++]

The following code example illustrates this process in C++.

```
// This example assumes that d3dDevice is a valid pointer to an
// IDirect3DDevice8 interface.
// lptexBaseTexture is a valid pointer to a texture.
// lptexSpecLightMap is a valid pointer to a texture that contains RGB
// specular light map data.
// lptexLightMap is a valid pointer to a texture that contains RGB
// light map data.

// Set the base texture.
d3dDevice->SetTexture(0, lptexBaseTexture );

// Set the base texture operation and arguments.
d3dDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// Set the specular light map.
d3dDevice->SetTexture(1, lptexSpecLightMap);

// Set the specular light map operation and args.
d3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADD );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );

// Set the RGB light map.
d3dDevice->SetTexture(2, lptexLightMap);
```

```
// Set the RGB light map operation and arguments.
d3dDevice->SetTextureStageState(2,D3DTSS_COLOROP, D3DTOP_MODULATE);
d3dDevice->SetTextureStageState(2,D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(2,D3DTSS_COLORARG2, D3DTA_CURRENT );
```

[Visual Basic]

The following Microsoft® Visual Basic® code example illustrates this process.

```
' This example assumes that d3dDevice is a valid reference to an
' Direct3DDevice8 object.
' texBaseTexture is a valid reference to a texture.
' texSpecLightMap is a valid reference to a texture that contains RGB
' specular light map data.
' texLightMap is a valid reference to a texture that contains RGB
' light map data.

' Set the base texture.
Call d3dDevice.SetTexture(0, texBaseTexture)

' Set the base texture operation and args.
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE)
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE)

' Set the specular light map.
Call d3dDevice.SetTexture(1, texSpecLightMap)

' Set the specular light map operation and args.
Call d3dDevice.SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADD)
Call d3dDevice.SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDevice.SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT)

' Set the RGB light map.
Call d3dDevice.SetTexture(2, texLightMap)

' Set the RGB light map operation and args
Call d3dDevice.SetTextureStageState(2, D3DTSS_COLOROP, D3DTOP_MODULATE)
Call d3dDevice.SetTextureStageState(2, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDevice.SetTextureStageState(2, D3DTSS_COLORARG2, D3DTA_CURRENT)
```

Diffuse Light Maps

When illuminated by a light source, matte surfaces display diffuse light reflection. The brightness of diffuse light depends on the distance from the light source and the angle between the surface normal and the light source direction vector. The diffuse lighting effects simulated by lighting calculations produce only general effects.

[C++]

Your application can simulate more complex diffuse lighting with texture light maps. Do this by adding the diffuse light map to the base texture, as shown in the following C++ code example.

```
// This example assumes that d3dDevice is a valid pointer to an
// IDirect3DDevice8 interface.
// lptexBaseTexture is a valid pointer to a texture.
// lptexDiffuseLightMap is a valid pointer to a texture that contains
// RGB diffuse light map data.

// Set the base texture.
d3dDevice->SetTexture(0,lptexBaseTexture );

// Set the base texture operation and args.
d3dDevice->SetTextureStageState(0,D3DTSS_COLOROP,
    D3DTOP_MODULATE );
d3dDevice->SetTextureStageState(0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// Set the diffuse light map.
d3dDevice->SetTexture(1,lptexDiffuseLightMap );

// Set the blend stage.
d3dDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
```

[Visual Basic]

Your application can simulate more complex diffuse lighting with texture light maps. Do this by adding the diffuse light map to the base texture, as shown in the following Microsoft® Visual Basic® code example.

```
' This example assumes that d3dDevice is a valid reference to
' a Direct3DDevice7 object.
' texBaseTexture is a valid reference to a texture.
' texDiffuseLightMap is a valid reference to a texture that contains
' RGB diffuse light map data.

' Set the base texture.
Call d3dDevice.SetTexture(0, texBaseTexture)

' Set the base texture operation and args.
```

```
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE)
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE)

' Set the diffuse light map.
Call d3dDevice.SetTexture(1, texDiffuseLightMap)

' Set the blend stage.
Call d3dDevice.SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE)
Call d3dDevice.SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDevice.SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT)
```

Compressed Texture Resources

Microsoft® Direct3D® provides services to compress surfaces that are used for texturing 3-D models. This section provides information on creating and manipulating the data in a compressed texture surface.
Information is contained in the following topics.

- What Are Compressed Textures?
- Compressed Texture Formats
- Using Compressed Textures

What Are Compressed Textures?

Texture maps are digitized images drawn on three-dimensional shapes to add visual detail. They are mapped onto these shapes during rasterization, and the process can consume large amounts of both the system bus and memory. To reduce the amount of memory consumed by textures, Microsoft® Direct3D® supports the compression of texture surfaces. Some Direct3D devices support compressed texture surfaces natively. On such devices, once you have created a compressed surface and loaded the data into it, the surface can be used in Direct3D like any other texture surface. Direct3D handles decompression when the texture is mapped to a 3-D object.

Compressed Texture Formats

This section contains information on the internal organization of compressed texture formats. You do not need these details to use compressed textures, because you can use D3DX functions for conversion to and from compressed formats. However, this information is useful if you want to operate on compressed surface data directly. Direct3D uses a compression format that divides texture maps into 4x4 texel blocks. If the texture contains no transparency—is opaque—or if the transparency is specified by a one-bit alpha, an 8-byte block represents the texture map block. If the texture map does contain transparent texels, using an alpha channel, a 16-byte block represents it.

- Opaque and One-Bit Alpha Textures
- Textures with Alpha Channels
- Understanding Storage Efficiency and Texture Compression
- Mixing Formats Within a Single Texture

Note

Any single texture must specify that its data is stored as 64 or 128 bits per group of 16 texels. If 64-bit blocks—that is, format DXT1—are used for the texture, it is possible to mix the opaque and one-bit alpha formats on a per-block basis within the same texture. In other words, the comparison of the unsigned integer magnitude of color_0 and color_1 is performed uniquely for each block of 16 texels.

When 128-bit blocks are used, the alpha channel must be specified in either explicit (format DXT2 or DXT3) or interpolated mode (format DXT4 or DXT5) for the entire texture. As with color, once interpolated mode is selected, either eight interpolated alphas or six interpolated alphas mode can be used on a block-by-block basis. Again the magnitude comparison of alpha_0 and alpha_1 is done uniquely on a block-by-block basis.

The pitch for DXTn formats is different from what was returned in DirectX 7. It now refers the pitch of a row of blocks. For example, if you have a width of 16, then you will have a pitch of four blocks (4*8 for DXT1, 4*16 for DXT2-5.)

Opaque and One-Bit Alpha Textures

Texture format DXT1 is for textures that are opaque or have a single transparent color.

For each opaque or one-bit alpha block, two 16-bit values (RGB 5:6:5 format) and a 4x4 bitmap with 2-bits-per-pixel are stored. This totals 64 bits for 16 texels, or four bits per texel. In the block bitmap, there are two bits per texel to select between the four colors, two of which are stored in the encoded data. The other two colors are derived from these stored colors by linear interpolation. This layout is shown in the following diagram.

The one-bit alpha format is distinguished from the opaque format by comparing the two 16-bit color values stored in the block. They are treated as unsigned integers. If the first color is greater than the second, it implies that only opaque texels are defined. This means four colors are used to represent the texels. In four-color encoding, there are two derived colors and all four colors are equally distributed in RGB color space. This format is analogous to RGB 5:6:5 format. Otherwise, for one-bit alpha transparency, three colors are used and the fourth is reserved to represent transparent texels.

In three-color encoding, there is one derived color and the fourth two-bit code is reserved to indicate a transparent texel (alpha information). This format is analogous to RGBA 5:5:5:1, where the final bit is used for encoding the alpha mask.

[C++]
The following code example illustrates the algorithm for deciding whether three- or four-color encoding is selected.

```
if (color_0 > color_1)
{
   // Four-color block: derive the other two colors.
   // 00 = color_0, 01 = color_1, 10 = color_2, 11 = color_3
   // These 2-bit codes correspond to the 2-bit fields
   // stored in the 64-bit block.
   color_2 = (2 * color_0 + color_1 + 1) / 3;
   color_3 = (color_0 + 2 * color_1 + 1) / 3;
}
else
{
   // Three-color block: derive the other color.
   // 00 = color_0,  01 = color_1,  10 = color_2,
   // 11 = transparent.
   // These 2-bit codes correspond to the 2-bit fields
   // stored in the 64-bit block.
    color_2 = (color_0 + color_1) / 2;
    color_3 = transparent;

}
```

It is recommended that you set the RGBA components of the transparency pixel to zero before blending.

The following tables show the memory layout for the 8-byte block. It is assumed that the first index corresponds to the y-coordinate and the second corresponds to the x-coordinate. For example, Texel[1][2] refers to the texture map pixel at (x,y) = (2,1). This table contains the memory layout for the 8-byte (64-bit) block.

Word address	**16-bit word**
0	Color_0
1	Color_1
2	Bitmap Word_0
3	Bitmap Word_1

Color_0 and Color_1—the colors at the two extremes—are laid out as follows:

Bits	**Color**
4:0 (LSB)	Blue color component
10:5	Green color component
15:11	Red color component

Bitmap Word_0 is laid out as follows:

Bits	Texel
1:0 (LSB)	Texel[0][0]
3:2	Texel[0][1]
5:4	Texel[0][2]
7:6	Texel[0][3]
9:8	Texel[1][0]
11:10	Texel[1][1]
13:12	Texel[1][2]
15:14 (MSB)	Texel[1][3]

Bitmap Word_1 is laid out as follows:

Bits	Texel
1:0 (LSB)	Texel[2][0]
3:2	Texel[2][1]
5:4	Texel[2][2]
7:6	Texel[2][3]
9:8	Texel[3][0]
11:10	Texel[3][1]
13:12	Texel[3][2]
15:14 (MSB)	Texel[3][3]

Example of Opaque Color Encoding

As an example of opaque encoding, assume that the colors red and black are at the extremes. Red is color_0, and black is color_1. There are four interpolated colors that form the uniformly distributed gradient between them. To determine the values for the 4x4 bitmap, the following calculations are used.

```
00 ? color_0
01 ? color_1
10 ? 2/3 color_0 + 1/3 color_1
11 ? 1/3 color_0 + 2/3 color_1
```

The bitmap then looks like the following graphic.

This looks like the following series of colors.

Example of One-Bit Alpha Encoding

This format is selected when the unsigned 16-bit integer, color_0, is less than the unsigned 16-bit integer, color_1. An example of where this format can be used is leaves on a tree, shown against a blue sky. Some texels can be marked as transparent while three shades of green are still available for the leaves. Two colors fix the extremes, and the third is an interpolated color.
Here is an example of such a picture.

Note that where the image is shown as white, the texel would be encoded as transparent. Also note that the RGBA components of the transparent texels should be set to zero before blending.
The bitmap encoding for the colors and the transparency is determined using the following calculations.

```
00 ? color_0
01 ? color_1
10 ? 1/2 color_0 + 1/2 color_1
11  ?  Transparent
```

The bitmap then looks like the following graphic.

Textures with Alpha Channels

There are two ways to encode texture maps that exhibit more complex transparency. In each case, a block that describes the transparency precedes the 64-bit block already described. The transparency is either represented as a 4x4 bitmap with four bits per pixel (explicit encoding), or with fewer bits and linear interpolation that is analogous to what is used for color encoding.
The transparency block and the color block are arranged as shown in the following table.

Word Address	**64-bit Block**
3:0	Transparency block
7:4	Previously described 64-bit block

Explicit Texture Encoding

For explicit texture encoding (DXT2 and DXT3 formats), the alpha components of the texels that describe transparency are encoded in a 4x4 bitmap with 4 bits per texel. These four bits can be achieved through a variety of means such as dithering or by simply using the four most significant bits of the alpha data. However they are produced, they are used just as they are, without any form of interpolation.
The following diagram shows a 64-bit transparency block.

Note

The compression method of Microsoft® Direct3D® uses the four most significant bits.

The following tables illustrate how the alpha information is laid out in memory, for each 16-bit word.

This table contains the layout for Word 0.

Bits	Alpha
3:0 (LSB)	[0][0]
7:4	[0][1]
11:8	[0][2]
15:12 (MSB)	[0][3]

This table contains the layout for Word 1.

Bits	Alpha
3:0 (LSB)	[1][0]
7:4	[1][1]
11:8	[1][2]
15:12 (MSB)	[1][3]

This table contains the layout for Word 2.

Bits	Alpha
3:0 (LSB)	[2][0]
7:4	[2][1]
11:8	[2][2]
15:12 (MSB)	[2][3]

This table contains the layout for Word 3.

Bits	Alpha
3:0 (LSB)	[3][0]
7:4	[3][1]
11:8	[3][2]
15:12 (MSB)	[3][3]

The difference between DXT2 and DXT3 is that in the DXT2 format it is assumed that the color data has been premultiplied by alpha. In the DXT3 format it is assumed the color is not premultiplied by alpha. These two formats are needed because in most cases by the time a texture is used, simply examining the data is not sufficient to determine if the color values have been multiplied by alpha. Because this information is needed at run time, the two FOURCC codes are used to differentiate these cases. However, the data and interpolation method used for these two formats is identical.

The color compare used in DXT1 to determine if the texel is transparent is not used in this format. It is assumed that without the color compare the color data is always treated as if in 4-color mode. In other words, the *if* statement at the top of the DXT1 code should be the following:

```
if ((color_0 > color_1) OR !DXT1) {
```

Three-Bit Linear Alpha Interpolation

The encoding of transparency for the DXT4 and DXT5 formats is based on a concept similar to the linear encoding used for color. Two 8-bit alpha values and a 4x4 bitmap with three bits per pixel are stored in the first eight bytes of the block. The representative alpha values are used to interpolate intermediate alpha values. Additional information is available in the way the two alpha values are stored. If alpha_0 is greater than alpha_1, then six intermediate alpha values are created by the interpolation. Otherwise, four intermediate alpha values are interpolated between the specified alpha extremes. The two additional implicit alpha values are 0 (fully transparent) and 255 (fully opaque).

[C++]

The following code example illustrates this algorithm.

```
// 8-alpha or 6-alpha block?
if (alpha_0 > alpha_1) {
   // 8-alpha block:  derive the other six alphas.
   // Bit code 000 = alpha_0, 001 = alpha_1, others are interpolated.
   alpha_2 = (6 * alpha_0 + 1 * alpha_1 + 3) / 7;    // bit code 010
   alpha_3 = (5 * alpha_0 + 2 * alpha_1 + 3) / 7;    // bit code 011
   alpha_4 = (4 * alpha_0 + 3 * alpha_1 + 3) / 7;    // bit code 100
   alpha_5 = (3 * alpha_0 + 4 * alpha_1 + 3) / 7;    // bit code 101
   alpha_6 = (2 * alpha_0 + 5 * alpha_1 + 3) / 7;    // bit code 110
   alpha_7 = (1 * alpha_0 + 6 * alpha_1 + 3) / 7;    // bit code 111
}
else {
   // 6-alpha block.
   // Bit code 000 = alpha_0, 001 = alpha_1, others are interpolated.
   alpha_2 = (4 * alpha_0 + 1 * alpha_1 + 2) / 5;    // Bit code 010
   alpha_3 = (3 * alpha_0 + 2 * alpha_1 + 2) / 5;    // Bit code 011
   alpha_4 = (2 * alpha_0 + 3 * alpha_1 + 2) / 5;    // Bit code 100
   alpha_5 = (1 * alpha_0 + 4 * alpha_1 + 2) / 5;    // Bit code 101
   alpha_6 = 0;                                      // Bit code 110
   alpha_7 = 255;                                    // Bit code 111
}
```

The memory layout of the alpha block is as follows:

Byte	Alpha
0	Alpha_0

1	Alpha_1
2	[0][2] (2 LSBs), [0][1], [0][0]
3	[1][1] (1 LSB), [1][0], [0][3], [0][2] (1 MSB)
4	[1][3], [1][2], [1][1] (2 MSBs)
5	[2][2] (2 LSBs), [2][1], [2][0]
6	[3][1] (1 LSB), [3][0], [2][3], [2][2] (1 MSB)
7	[3][3], [3][2], [3][1] (2 MSBs)

The difference between DXT4 and DXT5 is that in the DXT4 format it is assumed that the color data has been premultiplied by alpha. In the DXT5 format, it is assumed the color is not premultiplied by alpha. These two formats are needed because, in most cases, by the time a texture is used simply examining the data is not sufficient to determine if the color values have been multiplied by alpha. Because this information is needed at run time, the two FOURCC codes are used to differentiate these cases. However, the data and interpolation method used for these two formats is identical. The color compare used in DXT1 to determine if the texel is transparent is not used with these formats. It is assumed that without the color compare the color data is always treated as if in 4-color mode. In other words the *if* statement at the top of the DXT1 code should be:

```
if ((color_0 > color_1) OR !DXT1) {
```

Understanding Storage Efficiency and Texture Compression

All texture compression formats are powers of 2. While this does not mean that a texture is necessarily square, it does mean that both X and Y are powers of 2. For example, if a texture is originally 512×128 bytes, then the next mipmapping would be 256×64 and so on, with each level decreasing by a power of 2. At lower levels, where the texture is filtered to 16×2 and 8×1, there will be wasted bits because the compression block is always a 4×4 block of texels. Unused portions of the block are padded. Although there are wasted bits at the lowest levels, the overall gain is still significant. The worst case is, in theory, a 2K×1 texture (2^0 power). Here, only a single row of pixels is encoded per block, while the rest of the block is unused.

Mixing Formats Within a Single Texture

It is important to note that any single texture must specify that its data is stored as 64 or 128 bits per group of 16 texels. If 64-bit blocks—that is, format DXT1—are used for the texture, it is possible to mix the opaque and one-bit alpha formats on a per-block basis within the same texture. In other words, the comparison of the unsigned integer magnitude of color_0 and color_1 is performed uniquely for each block of 16 texels.

Once the 128-bit blocks are used, the alpha channel must be specified in either explicit (format DXT2 and DXT3) or interpolated mode (format DXT4 and DXT5) for the entire texture. As with color, once interpolated mode (format DXT4 and DXT5) is selected, then either eight interpolated alphas or six interpolated alphas

mode can be used on a block-by-block basis. Again the magnitude comparison of alpha_0 and alpha_1 is done uniquely on a block-by-block basis.

Using Compressed Textures

This section explains how to use compressed textures in a Microsoft® Direct3D® application.
Information is divided into the following topics.

- Determining Support for Compressed Textures
- Creating Compressed Textures
- Decompressing Compressed Textures

Determining Support for Compressed Textures

[C++]
Before your application creates a rendering device, it can determine if the device supports texturing from compressed texture surfaces by calling the **IDirect3D8::CheckDeviceFormat** method. This method determines whether a surface format can be used as a texture on a device representing the adapter. To test the adapter, specify any pixel format that uses the DXT1, DXT2, DXT3, DXT4, or DXT5 four character codes (FOURCCs). If **CheckDeviceFormat** returns D3D_OK, the device can create texture directly from a compressed texture surface that uses that format. If so, you can use compressed texture surfaces directly with Microsoft® Direct3D® by calling the **IDirect3DDevice8::SetTexture** method. The following code example shows how to determine if the adapter supports the D3DFMT_DXT1 compressed texture format.

```
BOOL IsCompressedTextureFormatOk( D3DFORMAT TextureFormat,
                                  D3DFORMAT AdapterFormat ) {
    HRESULT hr = pD3D->CheckDeviceFormat( D3DADAPTER_DEFAULT,
                                          D3DDEVTYPE_HAL,
                                          AdapterFormat,
                                          0,
                                          D3DRTYPE_TEXTURE,
                                          D3DFMT_DXT1);

    return SUCCEEDED( hr );
}
```

If the device does not support texturing from compressed texture surfaces, you can still store texture data in a compressed format surface, but you must convert any compressed textures to a supported format before they can be used for texturing.

[Visual Basic]
Before your application creates a rendering device, it can determine if the device supports texturing from compressed texture surfaces by calling **Direct3D8.CheckDeviceFormat**. This method determines whether a surface format

can be used as a texture on a device representing the adapter. To test the adapter, specify any pixel format that uses the DXT1, DXT2, DXT3, DXT4, or DXT5 four character codes (FOURCCs). If **CheckDeviceFormat** returns D3D_OK, the device can create texture directly from a compressed texture surface that uses that format. If so, you can use compressed texture surfaces directly with Microsoft® Direct3D® by calling the **Direct3DDevice8.SetTexture** method. If the device does not support texturing from compressed texture surfaces, you can still store texture data in a compressed format surface, but you must convert any compressed textures to a supported format before they can be used for texturing.

Creating Compressed Textures

[C++]
After creating a device that supports a compressed texture format on the adapter, you can create a compressed texture resource. Call **IDirect3DDevice8::CreateTexture** and specify a compressed texture format for the *Format* parameter.
Before loading an image into a texture object, retrieve a pointer to the texture surface by calling the **IDirect3DTexture8::GetSurfaceLevel** method.
Now you can use any D3DX function that begins with D3DXLoadSurface to load an image into the surface that was retrieved by using **GetSurfaceLevel**. These functions handle conversion to and from compressed texture formats.
You can create and convert compressed texture (DDS) files using the DXTex Tool supplied with the SDK. You can also create your own DDS files.

[Visual Basic]
After creating a device that supports a compressed texture format on the adapter, you can create a compressed texture resource. Call **Direct3DDevice8.CreateTexture** and specify a compressed texture format for the *Format* parameter.
Before loading an image into a texture object, retrieve a pointer to the texture surface by calling the **Direct3DTexture8.GetSurfaceLevel** method.
Now you can use any D3DX function that begins with D3DXLoadSurface to load an image into the surface that was retrieved by using **GetSurfaceLevel**. These functions handle conversion to and from compressed texture formats.
You can create and convert compressed texture (DDS) files using the DXTex Tool supplied with the SDK. You can also create your own DDS files.

The advantage of this behavior is that an application can copy the contents of a compressed surface to a file without calculating how much storage is required for a surface of a particular width and height in the specific format.
The following table shows the five types of compressed textures. For more information on how the data is stored, see Compressed Texture Formats. You only need this information if you are writing your own compression routines.

FOURCC	Description	Alpha-premultiplied?

DXT1	Opaque / one-bit alpha	N/A
DXT2	Explicit alpha	Yes
DXT3	Explicit alpha	No
DXT4	Interpolated alpha	Yes
DXT5	Interpolated alpha	No

Note
When you transfer data from a non-premultiplied format to a premultiplied format, Direct3D scales the colors based on the alpha values. Transferring data from a premultiplied format to a non-premultiplied format is not supported. If you try to transfer data from a premultiplied-alpha source to a non-premultiplied-alpha destination, the method returns D3DERR_INVALIDCALL. If you transfer data from a premultiplied-alpha source to a destination that has no alpha, the source color components, which have been scaled by alpha, are copied as is.

Decompressing Compressed Textures

As with compressing a texture surface, decompressing a compressed texture is performed through Microsoft® Direct3D® copying services.
If the driver supports the creation of compressed video-memory surfaces, then the driver can also decompress copies from a compressed video-memory surface to an uncompressed video- or system-memory surface.
Copies from compressed system-memory surfaces to uncompressed video-memory surfaces are largely unsupported and should not be attempted, even when the driver supports compressed textures. This does not mean that it is impossible to decompress a compressed system-memory surface and move its contents into a video-memory surface; it merely requires an additional step.

Û To decompress a system-memory surface into video memory

1. Create an uncompressed, off-screen plain surface in system memory of the desired dimensions and pixel format.
2. Copy from the compressed system-memory surface to the uncompressed system-memory surface.
3. Copy the uncompressed surface to the uncompressed video-memory surface.

Volume Texture Resources

Volume textures are discussed in the following topics.

- What Are Volume Textures?
- Using Volume Textures

What Are Volume Textures?

Volume textures are three-dimensional collections of pixels (texels) that can be used to paint a two-dimensional primitive such as a triangle or a line. Three-element texture coordinates are required for each vertex of a primitive that is to be textured with a volume. As the primitive is drawn, each contained pixel is filled with the color value from some pixel within the volume, in accordance with rules analogous to the two-dimensional texture case. Volumes are not rendered directly because there are no three-dimensional primitives that can be painted with them.
You can use volume textures for special effects such as patchy fog, explosions, and so on.
Volumes are organized into slices and can be thought of as *width* x *height* 2-D surfaces stacked to make a *width* x *height* x *depth* volume. Each slice is a single row. Volumes can have subsequent levels, in which the dimensions of each level are truncated to half the dimensions of the previous level. The illustration below gives an idea of what a volume texture with multiple levels looks like.

Using Volume Textures

[C++]
The code examples below show the steps required to use a volume texture.
First, specify a custom vertex type that has three texture coordinates for each vertex, as shown in this code example.

```
struct VOLUMEVERTEX
{
    FLOAT x, y, z;
    DWORD color;
    FLOAT tu, tv, tw;
};

#define D3DFVF_VOLUMEVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|
                    D3DFVF_TEX1|D3DFVF_TEXCOORDSIZE3(0))
```

Next, fill the vertices with data.

```
VOLUMEVERTEX g_vVertices[4] =
{
    { 1.0f, 1.0f, 0.0f, 0xffffffff, 1.0f, 1.0f, 0.0f },
    {-1.0f, 1.0f, 0.0f, 0xffffffff, 0.0f, 1.0f, 0.0f },
    { 1.0f,-1.0f, 0.0f, 0xffffffff, 1.0f, 0.0f, 0.0f },
    {-1.0f,-1.0f, 0.0f, 0xffffffff, 0.0f, 0.0f, 0.0f }
};
```

Now, create a vertex buffer and fill it with data from the vertices.
The next step is to use the **IDirect3DDevice8::CreateVolumeTexture** method to create a volume texture, as shown in this code example.

```
LPDIRECT3DVOLUMETEXTURE8 volTexture;
```

```
d3dDevice->CreateVolumeTexture( 8, 4, 4, 1, 0, D3DFMT_R8G8B8,
                        D3DPOOL_MANAGED, &volTexture );
```

Before rendering the primitive, set the current texture to the volume texture created above. The code example below shows the entire rendering process for a strip of triangles.

```
if( SUCCEEDED( d3dDevice->BeginScene() ) )
{
    // Draw the quad, with the volume texture.
    d3dDevice->SetTexture( 0, pVolumeTexture );
    d3dDevice->SetVertexShader( D3DFVF_VOLUMEVERTEX );
    d3dDevice->SetStreamSource( 0, pVB, sizeof(VOLUMEVERTEX) );
    d3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2);

   // End the scene.
   d3dDevice->EndScene();
}
```

[Visual Basic]

The code examples below show the steps required to use a volume texture.
First, specify a custom vertex type that has three texture coordinates for each vertex, as shown in this code example.

```
Private Type VOLUMEVERTEX
    Dim x As Single
    Dim y As Single
    Dim z As Single
    Dim color As Long
    Dim tu As Single
    Dim tv As Single
    Dim tw As Single
};

Const D3DFVF_VOLUMEVERTEX = (D3DFVF_XYZ Or D3DFVF_DIFFUSE Or _
                        D3DFVF_TEX1 Or D3DFVF_TEXCOORDSIZE3_0)
```

Next, fill the vertices with data.

```
Dim Vertices(3) As VOLUMEVERTEX

With Vertices(0):
   .x = 1.0: .y = 1.0: .z = 0.0: .color = &HFFFFFFFF .tu = 1.0: .tv = 1.0: .tw = 0.0
End With

With Vertices(1):
```

```
    .x = -1.0: .y = 1.0: .z = 0.0: .color = &HFFFFFFFF .tu = 0.0: .tv = 1.0: .tw = 0.0
End With

With Vertices(2):
    .x = 1.0: .y = -1.0: .z = 0.0: .color = &HFFFFFFFF .tu = 1.0: .tv = 0.0: .tw = 0.0
End With

With Vertices(3):
    .x = -1.0: .y = -1.0: .z = 0.0: .color = &HFFFFFFFF .tu = 0.0: .tv = 0.0: .tw = 0.0
End With
```

Now, create a vertex buffer and fill it with data from the vertices.
The next step is to use the **Direct3DDevice8.CreateVolumeTexture** method to create a volume texture, as shown in the code example below.

```
Dim volTexture As Direct3DVolumeTexture8

Set volTexture = m_D3DDevice.CreateVolumeTexture(8, 4, 4, 1, 0, _
                                    D3DFMT_R8G8B8, _
                                    D3DPOOL_MANAGED)
```

Before rendering the primitive, set the current texture to the volume texture created above. The code example below shows the entire rendering process for a strip of triangles.

```
d3dDevice.BeginScene

// Draw the quad, with the volume texture.
Call d3dDevice.SetTexture(0, volTexture)
Call d3dDevice.SetVertexShader(D3DFVF_VOLUMEVERTEX)
Call d3dDevice.SetStreamSource(0, VB, len(Vertices(0))
Call d3dDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2)

// End the scene.
d3dDevice->EndScene
```

Automatic Texture Management

Texture management is the process of determining which textures are needed for rendering at a given time and ensuring that those textures are loaded into video memory. As with any algorithm, texture management schemes vary in complexity, but any approach to texture management involves the following key tasks.

- Tracking the amount of available texture memory.
- Calculating which textures are needed for rendering, and which are not.

- Determining which existing texture resources can be reloaded with another texture image, and which surfaces should be destroyed and replaced with new texture resources.

Microsoft® Direct3D® implements system-supported texture management to ensure that textures are loaded for optimal performance. Texture resources that Direct3D manages are referred to as *managed textures*.
The texture manager tracks textures with a time-stamp that identifies when the texture was last used. It then uses a least-recently-used algorithm to determine which textures should be removed. Texture priorities are used as tie breakers when two textures are targeted for removal from memory. If two textures have the same priority value, the least-recently-used texture is removed. However, if two textures have the same time-stamp, the texture that has a lower priority is removed first.

[C++]
You request automatic texture management for texture surfaces when you create them. To retrieve a managed texture in a C++ application, create a texture resource by calling **IDirect3DDevice8::CreateTexture** and specifying the D3DPOOL_MANAGED for the *Pool* parameter. You are not allowed to specify where you want the texture created. You cannot use the D3DPOOL_DEFAULT or D3DPOOL_SYSTEMMEM flags when creating a managed texture. After creating the managed texture, you can call the **IDirect3DDevice8::SetTexture** method to set it to a stage in the rendering device's texture cascade.
You can assign a priority to managed textures by calling the **IDirect3DResource8::SetPriority** method for the texture surface.

[Visual Basic]
You request automatic texture management for texture surfaces when you create them. To retrieve a managed texture in a Microsoft® Visual Basic® application, create a texture resource by calling **Direct3DDevice8.CreateTexture** and specifying the D3DPOOL_MANAGED for the *Pool* parameter. You are not allowed to specify where you want the texture created. You cannot use the D3DPOOL_DEFAULT or D3DPOOL_SYSTEMMEM flags when creating a managed texture. After creating the managed texture, you can call the **Direct3DDevice8.SetTexture** method to set it to a stage in the rendering device's texture cascade.
You can assign a priority to managed textures by calling the **Direct3DResource8.SetPriority** method for the texture surface.

Direct3D automatically downloads textures into video memory as needed. The system might cache managed textures in local or nonlocal video memory, depending on the availability of nonlocal video memory or other factors. The cache location of your managed textures is not communicated to your application, nor is this information required to use automatic texture management. If your application uses more textures than can fit in video memory, Direct3D removes older textures from video memory to make room for the new textures. If you use a removed texture again, the system uses the original system-memory texture surface to reload the texture in the video-memory

cache. Although reloading the texture is necessary, it also decreases the application's performance.
You can dynamically modify the original system-memory copy of the texture by updating or locking the texture resource. When the system detects a dirty surface—after an update is completed, or when the surface is unlocked—the texture manager automatically updates the video-memory copy of the texture. The performance hit incurred is similar to reloading a removed texture.

[C++]
When entering a new level in a game, your application might need to flush all managed textures from video memory. You can explicitly request to remove all managed textures by calling the **IDirect3DDevice8::ResourceManagerDiscardBytes** method and specifying a value of 0 for the *Bytes* parameter. When you call this method, Direct3D destroys any cached local and nonlocal video-memory textures, but leaves the original system-memory copies untouched.

[Visual Basic]
When entering a new level in a game, your application may need to flush all managed textures from video memory. You can explicitly request to remove all managed textures by calling the **Direct3DDevice8.ResourceManagerDiscardBytes** method and specifying a value of 0 for the *Bytes* parameter. When you call this method, Direct3D destroys any cached local and nonlocal video-memory textures, but leaves the original system-memory copies untouched.

For more information on resource management, see Managing Resources.

Hardware Considerations for Texturing

Current hardware does not necessarily implement all the functionality that the Microsoft® Direct3D® interface enables. Your application must test user hardware and adjust its rendering strategies accordingly.
Many 3-D accelerator cards do not support diffuse iterated values as arguments to blending units. However, your application can introduce iterated color data when it performs texture blending.
Some 3-D hardware may not have a blending stage associated with the first texture. On these adapters, your application must perform blending in the second and third texture stages in the set of current textures.

[C++]
Due to limitations in much of today's hardware, few display adapters can perform trilinear mipmap interpolation through the multiple texture blending interface offered by **IDirect3DDevice8**. Specifically, there is little support for setting the D3DTSS_MIPFILTER texture stage state to D3DTEXF_LINEAR. Your application can use multipass texture blending to achieve the same effects, or degrade to the D3DTEXF_POINT mipmap filter mode, which is widely supported.

[Visual Basic]
Due to limitations in much of today's hardware, few display adapters can perform trilinear mipmap interpolation through the multiple texture blending interface offered by **Direct3DDevice8**. Specifically, there is little support for setting the D3DTSS_MIPFILTER texture stage state to D3DTEXF_LINEAR. Your application can use multipass texture blending to achieve the same effects, or degrade to the D3DTEXF_POINT mipmap filter mode, which is widely supported.

Depth Buffers

This section presents information on using depth buffers for hidden line and hidden surface removal. It is organized into the following topics.

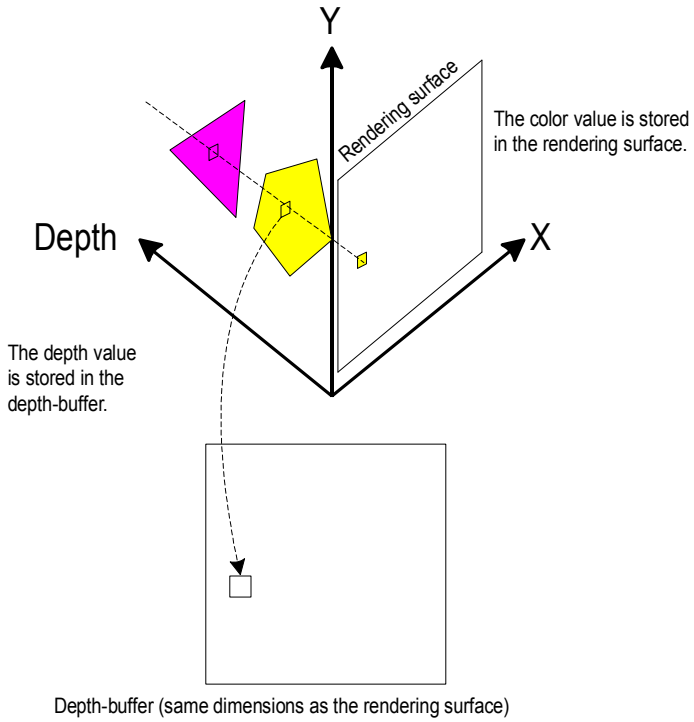
- What Are Depth Buffers?
- Using Depth Buffers

What Are Depth Buffers?

A depth buffer, often called a z-buffer or a w-buffer, is a property of the device that stores depth information to be used by Microsoft® Direct3D®. When Direct3D renders a 3-D scene to a target surface, it can use the memory in an associated depth-buffer surface as a workspace to determine how the pixels of rasterized polygons occlude one another. Direct3D uses an off-screen Direct3D surface as the target to which final color values are written. The depth-buffer surface that is associated with the render-target surface is used to store depth information that tells Direct3D how deep each visible pixel is in the scene.

When a 3-D scene is rasterized with depth buffering enabled, each point on the rendering surface is tested. The values in the depth buffer can be a point's z-coordinate or its homogeneous w-coordinate—from the point's (x,y,z,w) location in projection space. A depth buffer that uses z values is often called a z-buffer, and one that uses w values is called a w-buffer. Each type of depth buffer has advantages and disadvantages, which are discussed later.

At the beginning of the test, the depth value in the depth buffer is set to the largest possible value for the scene. The color value on the rendering surface is set to either the background color value or the color value of the background texture at that point. Each polygon in the scene is tested to see if it intersects with the current coordinate (x,y) on the rendering surface. If it does, the depth value—which will be the z coordinate in a z-buffer, and the w coordinate in a w-buffer—at the current point is tested to see if it is smaller than the depth value stored in the depth buffer. If the depth of the polygon value is smaller, it is stored in the depth buffer and the color value from the polygon is written to the current point on the rendering surface. If the depth value of the polygon at that point is larger, the next polygon in the list is tested. This process is shown in the following illustration.



[C++]

Note

Although most applications don't use this feature, you can change the comparison that Direct3D uses to determine which values are placed in the depth buffer and subsequently the render-target surface. To do so, change the value for the D3DRS_ZFUNC render state.

[Visual Basic]

Note

Although most applications don't use this feature, you can change the comparison Direct3D uses to determine which values are placed in the depth buffer and subsequently the render-target surface. To do so, change the value for the D3DRS_ZFUNC render state.

Nearly all 3-D accelerators on the market support z-buffering, making z-buffers the most common type of depth buffer today. However ubiquitous, z-buffers have their drawbacks. Due to the mathematics involved, the generated z values in a z-buffer tend not to be distributed evenly across the z-buffer range (typically 0.0 to 1.0, inclusive). Specifically, the ratio between the far and near clipping planes strongly affects how

unevenly z values are distributed. Using a far-plane distance to near-plane distance ratio of 100, 90% of the depth buffer range is spent on the first 10% of the scene depth range. Typical applications for entertainment or visual simulations with exterior scenes often require far-plane/near-plane ratios of anywhere between 1000 to 10000. At a ratio of 1000, 98% of the range is spent on the 1st 2% of the depth range, and the distribution becomes worse with higher ratios. This can cause hidden surface artifacts in distant objects, especially when using 16-bit depth buffers, the most commonly supported bit-depth.
A w-based depth buffer, on the other hand, is often more evenly distributed between the near and far clip planes than a z-buffer. The key benefit is that the ratio of distances for the far and near clipping planes is no longer an issue. This allows applications to support large maximum ranges, while still getting relatively accurate depth buffering close to the eye point. A w-based depth buffer isn't perfect, and can sometimes exhibit hidden surface artifacts for near objects. Another drawback to the w-buffered approach is related to hardware support: w-buffering isn't supported as widely in hardware as z-buffering.
Z-buffering requires overhead during rendering. Various techniques can be used to optimized rendering when using z-buffers. For details, see Z-Buffer Performance.

Note
The actual interpretation of a depth value is specific to the 3-D renderer.

Using Depth Buffers

The following topics discuss common usage scenarios for depth buffers.

- Querying for Depth Buffer Support
- Creating a Depth Buffer
- Enabling Depth Buffering
- Retrieving a Depth Buffer
- Clearing Depth Buffers
- Changing Depth Buffer Write Access
- Changing Depth Buffer Comparison Functions
- Using Z-Bias

See Also

What Are Depth Buffers?

Querying for Depth Buffer Support

As with any feature, the driver that your application uses might not support all types of depth buffering. Always check the driver's capabilities. Although most drivers support z-based depth buffering, not all will support w-based depth buffering. Drivers do not fail if you attempt to enable an unsupported scheme. They fall back on another

depth buffering method instead, or sometimes disable depth buffering altogether, which can result in scenes rendered with extreme depth-sorting artifacts.
You can check for general support for depth buffers by querying Microsoft® Direct3D® for the display device that your application will use before you create a Direct3D device. If the Direct3D object reports that it supports depth buffering, any hardware devices you create from this Direct3D object will support z-buffering.

[C++]

Û To query for general depth buffering support in a C++ application

To query for depth buffering support, you can use the **IDirect3D8::CheckDeviceFormat** method, as shown in the following code example.

```
// The following example assumes that pCaps is a valid pointer to an
// initialized D3DCAPS8 structure.
   if( FAILED( m_pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal,
                                          pCaps->DeviceType, Format,
                                          D3DUSAGE_DEPTHSTENCIL,
                                          D3DRTYPE_SURFACE,
                                          D3DFMT_D16 ) ) )
      return E_FAIL;
```

CheckDeviceFormat allows you to choose a device to create based on the capabilities of that device. In this case, devices that do not support 16-bit depth buffers are rejected.
In addition, you can use **IDirect3D8::CheckDepthStencilMatch** to determine whether a supported depth-stencil format is compatible with the render target format in the display mode, particularly if a depth format must be equal to the render target format.
Using **CheckDepthStencilMatch** to determine depth-stencil compatibility with a render target is illustrated in the following code example.

```
// Reject devices that cannot create a render target of RTFormat while
// the back buffer is of RTFormat and the depth-stencil buffer is
// at least 8 bits of stencil.
if( FAILED( m_pD3D->CheckDepthStencilMatch( pCaps->AdapterOrdinal,
                                            pCaps->DeviceType,
                                            AdapterFormat,
                                            RTFormat,
                                            D3DFMT_D24S8 ) ) )
      return E_FAIL;
```

[Visual Basic]

Û To query for general depth buffering support in a Visual Basic application

To query for depth buffering support, you can use the **Direct3D8.CheckDeviceFormat** method, as shown in the following code example.

```
' The following example assumes that g_D3D, AdapterOrdinal,
' DeviceType, and Format are properly initialized values.
g_D3D.CheckDeviceFormat AdapterOrdinal, DeviceType, Format, _
            D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE, _
            D3DFMT_D16
```

CheckDeviceFormat allows you to choose a device to create based on the capabilities of that device. In this case, devices that do not support 16-bit depth buffers are rejected.
In addition, you can use **Direct3D8.CheckDepthStencilMatch** to determine whether a supported depth-stencil format is compatible with the render target format in the display mode, particularly if a depth format must be equal to the render target format.

Once you know that the driver supports depth buffers, you can verify w-buffer support. Although depth buffers are supported for all software rasterizers, w-buffers are supported only by the reference rasterizer, which is not suited for use by real-world applications. Regardless of the type of device your application uses, verify support for w-buffers before you attempt to enable w-based depth buffering.

[C++]

To determine support for w-buffers in a C++ application

1. After creating your device, call the **IDirect3DDevice8::GetDeviceCaps** method, passing an initialized **D3DCAPS8** structure.
2. After the call, the **LineCaps** member contains information about the driver's support for rendering primitives.
3. If the **RasterCaps** member of this structure contains the D3DPRASTERCAPS_WBUFFER flag, then the driver supports w-based depth buffering for that primitive type.

[Visual Basic]

To determine support for w-buffers in a Visual Basic application

1. After creating your device, call the **Direct3DDevice8.GetDeviceCaps** method, passing an initialized **D3DCAPS8** type.
2. After the call, the **LineCaps** member contains information about the driver's support for rendering primitives.
3. If the **RasterCaps** member of this type contains the D3DPRASTERCAPS_WBUFFER flag, then the driver supports w-based depth buffering for that primitive type.

For general information about depth buffering, see What Are Depth Buffers?

Creating a Depth Buffer

[C++]
A depth buffer is a property of the device. To create a depth buffer that is managed by Microsoft® Direct3D®, set the appropriate members of the **D3DPRESENT_PARAMETERS** structure as shown in the following code example.

```
D3DPRESENT_PARAMETERS d3dpp;

ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed               = TRUE;
d3dpp.SwapEffect             = D3DSWAPEFFECT_COPY_VSYNC;
d3dpp.EnableAutoDepthStencil = TRUE;
d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
```

By setting the **EnableAutoDepthStencil** member to TRUE, you instruct Direct3D to manage depth buffers for the application. Note that **AutoDepthStencilFormat** must be set to a valid depth buffer format. The D3DFMT_D16 flag specifies a 16-bit depth buffer, if one is available.
The following call to the **IDirect3D8::CreateDevice** method creates a device that then creates a depth buffer.

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                  &d3dpp, &d3dDevice ) ) )
    return E_FAIL;
```

The depth buffer is automatically set as the render target of the device. When the device is reset, the depth buffer is automatically destroyed and recreated in the new size.
To create a new depth buffer surface, use the **IDirect3DDevice8::CreateDepthStencilSurface** method.
To set a new depth-buffer surface for the device, use the **IDirect3DDevice8::SetRenderTarget** method.
To use the depth buffer in your application, you need to enable the depth buffer. For details, see Enabling Depth Buffering.

[Visual Basic]
A depth buffer is a property of the device. To create a depth buffer that is managed by Microsoft® Direct3D®, set the appropriate members of the **D3DPRESENT_PARAMETERS** structure as shown in the following code example.

```
Dim d3dpp As D3DPRESENT_PARAMETERS

d3dpp.Windowed = 1
d3dpp.SwapEffect = D3DSWAPEFFECT_COPY_VSYNC
d3dpp.EnableAutoDepthStencil = 1
```

```
d3dpp.AutoDepthStencilFormat = D3DFMT_D16
```

By setting the **EnableAutoDepthStencil** member to a nonzero value (True), you instruct Direct3D to manage depth buffers for the application. Note that **AutoDepthStencilFormat** must be set to a valid depth buffer format. The D3DFMT_D16 flag specifies a 16-bit depth buffer, if one is available.
The following call to the **Direct3D8.CreateDevice** method creates a device that then creates a depth buffer.

```
Set d3dDevice = g_D3D.CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
hWnd, _
                        D3DCREATE_SOFTWARE_VERTEXPROCESSING, d3dpp)
If d3dDevice Is Nothing Then Exit Function
```

The depth buffer is automatically set as the render target of the device. When the device is reset, the depth buffer is automatically destroyed and recreated in the new size.
To create a new depth buffer surface, use the **Direct3DDevice8.CreateDepthStencilSurface** method.
To set a new depth-buffer surface for the device, use the **Direct3DDevice8.SetRenderTarget** method.
To use the depth buffer in your application, you must enable the depth buffer. For details, see Enabling Depth Buffering.

Enabling Depth Buffering

[C++]
After you create a depth buffer, as described in Creating a Depth Buffer, you enable depth buffering by calling the **IDirect3DDevice8::SetRenderState** method. Set the D3DRS_ZENABLE render state to enable depth-buffering. Use the D3DZB_TRUE member of the **D3DZBUFFERTYPE** enumerated type (or TRUE) to enable z-buffering, D3DZB_USEW to enable w-buffering, or D3DZB_FALSE (or FALSE) to disable depth buffering.

Note

To use w-buffering, your application must set a compliant projection matrix even if it doesn't use the Microsoft® Direct3D® transformation pipeline. For information about providing an appropriate projection matrix, see A W-Friendly Projection Matrix. The projection matrix discussed in What Is the Projection Transformation? is compliant.

[Visual Basic]
After you create a depth buffer, as described in Creating a Depth Buffer, you enable depth buffering by calling the **Direct3DDevice8.SetRenderState** method. Set the D3DRS_ZENABLE render state to enable depth-buffering. Use the D3DZB_TRUE

member of the **CONST_D3DZBUFFERTYPE** enumeration to enable z-buffering, D3DZB_USEW to enable w-buffering, or D3DZB_FALSE to disable depth buffering altogether.

Note

To use w-buffering, your application must set a compliant projection matrix, even if it doesn't use the Microsoft® Direct3D® transformation pipeline. For information about providing an appropriate projection matrix, see A W-Friendly Projection Matrix. The projection matrix discussed in What Is the Projection Transformation? is compliant.

Retrieving a Depth Buffer

[C++]
The following code example shows how to use the **IDirect3DDevice8::GetDepthStencilSurface**method to retrieve a pointer to the depth-buffer surface owned by the device.

```
LPDIRECT3DSURFACE8 pZBuffer;

m_d3dDevice->GetDepthStencilSurface( &pZBuffer );
```

[Visual Basic]
The following code example shows how to use the **Direct3DDevice8.GetDepthStencilSurface** method to retrieve the depth buffer-surface owned by the device.

```
Dim ZBuffer As Direct3DSurface8

Set ZBuffer = d3dDevice.GetDepthStencilSurface
```

Clearing Depth Buffers

[C++]
Many C++ applications clear the depth buffer before rendering each new frame. You can explicitly clear the depth buffer through Microsoft® Direct3D® by calling **IDirect3DDevice8::Clear** and specifying D3DCLEAR_ZBUFFER for the *Flags* parameter. The **Clear** method allows you to specify an arbitrary depth value in the *Z* parameter.

[Visual Basic]

Many Microsoft® Visual Basic® applications clear the depth buffer before rendering each new frame. You can explicitly clear the depth buffer through Microsoft® Direct3D® by calling **Direct3DDevice8.Clear** and specifying D3DCLEAR_ZBUFFER for the *Flags* parameter. The **Clear** method allows you to specify an arbitrary depth value in the *Z* parameter.

Changing Depth Buffer Write Access

By default, the Microsoft® Direct3D® system is allowed to write to the depth buffer. Most applications leave writing to the depth buffer enabled, but you can achieve some special effects by not allowing the Direct3D system to write to the depth buffer.

[C++]
You can disable depth buffer writes in C++ by calling the **IDirect3DDevice8::SetRenderState** method with the *State* parameter set to D3DRS_ZWRITEENABLE and the *Value* parameter set to 0.

[Visual Basic]
You can disable depth buffer writes in a Microsoft Visual Basic® application by calling the **Direct3DDevice8.SetRenderState** method with the *State* parameter set to D3DRS_ZWRITEENABLE and the *Value* parameter set to 0.

Changing Depth Buffer Comparison Functions

[C++]
By default, when depth-testing is performed on a rendering surface, the Microsoft® Direct3D® system updates the render-target surface if the corresponding depth value (z or w) for each point is less than the value in the depth buffer. In a C++ application, you change how the system performs comparisons on depth values by calling the **IDirect3DDevice8::SetRenderState** method with the *State* parameter set to D3DRS_ZFUNC. The *Value* parameter should be set to a value in the **D3DCMPFUNC** enumerated type.

[Visual Basic]
By default, when depth testing is performed on a rendering surface, the Microsoft® Direct3D® system updates the render-target surface if the corresponding depth value (z or w) for each point is less than the value in the depth buffer. In Microsoft® Visual Basic®, you can change how the system performs comparisons on depth values by calling the **Direct3DDevice8.SetRenderState** method with the *State* parameter set to D3DRS_ZFUNC. The *Value* parameter should be set to a value in the **CONST_D3DCMPFUNC** enumeration.

Using Z-Bias

Polygons that are coplanar in your 3-D space can be made to appear as if they are not coplanar by adding a z-bias to each one. This is a technique commonly used to ensure that shadows in a scene are displayed properly. For instance, a shadow on a wall will likely have the same depth value as the wall does. If you render the wall first and then the shadow, the shadow might not be visible, or depth artifacts might be visible. You can reverse the order in which you render the coplanar objects in hopes of reversing the effect, but depth artifacts are still likely.

[C++]
A C++ application can help ensure that coplanar polygons are rendered properly by adding a bias to the z-values that the system uses when rendering the sets of coplanar polygons. To add a z-bias to a set of polygons, call the **IDirect3DDevice8::SetRenderState** method just before rendering them, setting the *State* parameter to D3DRS_ZBIAS, and the *Value* parameter to a value between 0-16 inclusive. A higher z-bias value increases the likelihood that the polygons you render will be visible when displayed with other coplanar polygons.

[Visual Basic]
A Microsoft® Visual Basic® application can help ensure that coplanar polygons are rendered properly by adding a bias to the z-values that the system uses when rendering the sets of coplanar polygons. To add a z-bias to a set of polygons, call the **Direct3DDevice8.SetRenderState** method just before rendering them, setting the *State* parameter to D3DRS_ZBIAS, and the *Value* parameter to a value between 0-16 inclusive. A higher z-bias value increases the likelihood that the polygons you render will be visible when displayed with other coplanar polygons.

Stencil Buffers

This section presents a discussion of the purpose and use stencil buffers. It is divided into the following topics.

- What Is a Stencil Buffer?
- How the Stencil Buffer Works
- Customizing the Stencil Buffer

What Is a Stencil Buffer?

The stencil buffer enables or disables drawing to the rendering target surface on a pixel-by-pixel basis. At its most fundamental level, it enables applications to mask sections of the rendered image so that they are not displayed. Applications often use stencil buffers for special effects such as dissolves, decaling, and outlining. For details, see Stencil Buffer Techniques.

[C++]

Stencil buffer information is embedded in the z-buffer data. Your application can use the **IDirect3D8::CheckDeviceFormat** method to check for hardware stencil support, as shown in the following code example.

```
// Reject devices that cannot perform 8-bit stencil buffering.
// The following example assumes that pCaps is a valid pointer
// to an initialized D3DCAPS8 structure.

if( FAILED( m_pD3D->CheckDeviceFormat( pCaps->AdapterOrdinal,
                                       pCaps->DeviceType,
                                       Format,
                                       D3DUSAGE_DEPTHSTENCIL,
                                       D3DRTYPE_SURFACE,
                                       D3DFMT_D24S8 ) ) )
    return E_FAIL;
```

CheckDeviceFormat allows you to choose a device to create based on the capabilities of that device. In this case, devices that do not support 8-bit stencil buffers are rejected. Note that this is only one possible use for **CheckDeviceFormat**; for details see Determining Hardware Support.
To determine the stencil buffer limitations of a device, query the **StencilCaps** member of the **D3DCAPS8** structure for its supported stencil buffer operations.

[Visual Basic]
Stencil buffer information is embedded in the z-buffer data. Your application can use **Direct3D8.CheckDeviceFormat** to check for hardware stencil support, as shown in the following code example.

```
' The following example assumes that g_D3D, AdapterOrdinal,
' DeviceType, and Format are properly initialized values.

g_D3D.CheckDeviceFormat AdapterOrdinal, DeviceType, Format, _
                D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE, _
                D3DFMT_D24S8
If Err.Number <> D3D_OK Then
   ' Handle the error.
End If
```

CheckDeviceFormat allows you to choose a device to create based on the capabilities of that device. In this case, devices that do not support 8-bit stencil buffers set **Err.Number** to an error value. Note that this is only one possible use for **CheckDeviceFormat**; for details see Determining Hardware Support.
To determine the stencil buffer limitations of a device, query the **StencilCaps** member of the **D3DCAPS8** type for its supported stencil buffer operations.

How the Stencil Buffer Works

[C++]
Microsoft® Direct3D® performs a test on the contents of the stencil buffer on a pixel-by-pixel basis. For each pixel in the target surface, it performs a test using the corresponding value in the stencil buffer, a stencil reference value, and a stencil mask value. If the test passes, Direct3D performs an action. The test is performed using the following steps.

1. Perform a bitwise **AND** operation of the stencil reference value with the stencil mask.
2. Perform a bitwise **AND** operation of the stencil buffer value for the current pixel with the stencil mask.
3. Compare the result of step 1 to the result of step 2, using the comparison function.

These steps are shown in the following code example.

```
(StencilRef & StencilMask) CompFunc (StencilBufferValue & StencilMask)
```

StencilBufferValue is the contents of the stencil buffer for the current pixel. This code example uses the ampersand (&) symbol to represent the bitwise **AND** operation. *StencilMask* represents the value of the stencil mask, and *StencilRef* represents the stencil reference value. *CompFunc* is the comparison function.
The current pixel is written to the target surface if the stencil test passes, and is ignored otherwise. The default comparison behavior is to write the pixel, no matter how each bitwise operation turns out (D3DCMP_ALWAYS). You can change this behavior by changing the value of the D3DRS_STENCILFUNC render state, passing a member of the **D3DCMPFUNC** enumerated type to identify the desired comparison function.

[Visual Basic]
Microsoft® Direct3D® performs a test on the contents of the stencil buffer on a pixel-by-pixel basis. For each pixel in the target surface, it performs a test using the corresponding value in the stencil buffer, a stencil reference value, and a stencil mask value. If the test passes, Direct3D performs an action. The test is performed using the following steps.

1. Perform a bitwise **And** operation of the stencil reference value with the stencil mask.
2. Perform a bitwise **And** operation of the stencil buffer value for the current pixel with the stencil mask.
3. Compare the result of step 1 to the result of step 2, using the comparison function.

These steps are shown in the following code example.

```
(StencilRef And StencilMask) CompFunc (StencilBufferValue And StencilMask)
```

StencilBufferValue is the contents of the stencil buffer for the current pixel. *StencilMask* represents the value of the stencil mask, and *StencilRef* represents the stencil reference value. *CompFunc* is the comparison function.
The current pixel is written to the target surface if the stencil test passes, and is ignored otherwise. The default comparison behavior is to write the pixel, no matter how each bitwise operation turns out (D3DCMP_ALWAYS). You can change this behavior by changing the value of the D3DRS_STENCILFUNC render state, passing a member of the **CONST_D3DCMPFUNC** enumeration to identify the desired comparison function.

Customizing the Stencil Buffer

Your application can customize the operation of the stencil buffer. It can set the comparison function, the stencil mask, and the stencil reference value. It can also control the action that Microsoft® Direct3D® takes when the stencil test passes or fails. For more information, see Stencil Buffer State.

Vertex Buffers

This section introduces the concepts necessary to understand and use vertex buffers in a Microsoft® Direct3D® application. Information is divided into the following sections.

- What Are Vertex Buffers?
- Vertex Buffer Descriptions
- Pool and Usage for Vertex Buffers
- FVF Vertex Buffers
- Device Types and Vertex Processing Requirements
- Using Vertex Buffers

What Are Vertex Buffers?

[C++]
Vertex buffers, represented by the **IDirect3DVertexBuffer8** interface, are memory buffers that contain vertex data. Vertex buffers can contain any vertex type—transformed or untransformed, lit or unlit—that can be rendered through the use of the rendering methods in the **IDirect3DDevice8** interface. You can process the vertices in a vertex buffer to perform operations such as transformation, lighting, or generating clipping flags. Transformation is always performed.

[Visual Basic]
Vertex buffers, represented by the **Direct3DVertexBuffer8** class, are memory buffers that contain vertex data. Vertex buffers can contain any vertex type—transformed or

untransformed, lit or unlit—that can be rendered through use of the rendering methods in the **Direct3DDevice8** class. You can process the vertices in a vertex buffer to perform operations such as transformation, lighting, or generating clipping flags. Transformation is always performed.

The flexibility of vertex buffers make them ideal staging points for reusing transformed geometry. You could create a single vertex buffer, transform, light, and clip the vertices in it, and render the model in the scene as many times as needed without re-transforming it, even with interleaved render state changes. This is useful when rendering models that use multiple textures: the geometry is transformed only once, and then portions of it can be rendered as needed, interleaved with the required texture changes. Render state changes made after vertices are processed take effect the next time the vertices are processed. For more information, see Processing Vertices.

Vertex Buffer Descriptions

[C++]
A vertex buffer is described in terms of its capabilities: if it can exist only in system memory, if it is only used for write operations, and the type and number of vertices it can contain. All these traits are held in a **D3DVERTEXBUFFER_DESC** structure.

[Visual Basic]
A vertex buffer is described in terms of its capabilities: if it can exist only in system memory, if it is only used for write operations, and the type and number of vertices it can contain. All these traits are held in a **D3DVERTEXBUFFER_DESC** type.

Vertex buffer descriptions tell your application how an existing buffer was created and if it has been optimized since being created. You provide an empty description structure for the system to fill with the capabilities of a previously created vertex buffer. For more information about this task, see Retrieving Vertex Buffer Descriptions.

[C++]
The **Format** member is set to D3DFMT_VERTEXDATA to indicate that this is a vertex buffer. The **Type** identifies the resource type of the vertex buffer. The **Usage** structure member contains general capability flags. The D3DUSAGE_SOFTWAREPROCESSING flag indicates that the vertex buffer is to be used with software vertex processing. The presence of the D3DUSAGE_WRITEONLY flag in **Usage** indicates that the vertex buffer memory is used only for write operations. This frees the driver to place the vertex data in the best memory location to enable fast processing and rendering. If the D3DUSAGE_WRITEONLY flag is not used, the driver is less likely to put the data in a location that is inefficient for read operations. This sacrifices some processing

and rendering speed. If this flag is not specified, it is assumed that applications perform read and write operations on the data within the vertex buffer.
Pool specifies the memory class that is allocated for the vertex buffer. The D3DPOOL_SYSTEMMEM flag indicates that the system created the vertex buffer in system memory.
The **Size** member simply stores the size, in bytes, of the vertex buffer data. The **FVF** member contains a combination of Flexible Vertex Format Flags that identify the type of vertices that the buffer contains.

[Visual Basic]
The **format** member is used to describe the surface format of the vertex buffer data. The **type** identifies the resource type of the vertex buffer. The **Usage** structure member contains general capability flags. The D3DUSAGE_SOFTWAREPROCESSING flag indicates that the vertex buffer is to be used with software vertex processing. The presence of the D3DUSAGE_WRITEONLY flag in **Usage** indicates that the vertex buffer memory is used only for write operations. This frees the driver to place the vertex data in the best memory location to enable fast processing and rendering. If the D3DUSAGE_WRITEONLY flag is not used, the driver is less likely to put the data in a location that is inefficient for read operations. This sacrifices some processing and rendering speed. If this flag is not specified, it is assumed that applications perform read and write operations on the data within the vertex buffer.
Pool specifies the memory class that is allocated for the vertex buffer. The D3DPOOL_SYSTEMMEM flag indicates that the system created the vertex buffer in system memory.
The **size** member simply stores the size, in bytes, of the vertex buffer data. The **FVF** member contains a combination of Flexible Vertex Format Flags that identify the type of vertices that the buffer contains.

Pool and Usage for Vertex Buffers

[C++]
You can create vertex buffers with the **IDirect3DDevice8::CreateVertexBuffer** method, which takes pool (memory class) and usage parameters.
CreateVertexBuffer can also be created with a specified flexible vertex format (FVF) code for use in fixed function vertex processing, or as the output of process vertices. For details, see FVF Vertex Buffers and FVF Usage Settings for Destination Vertex Buffers.

[Visual Basic]
You can create vertex buffers with the **Direct3DDevice8.CreateVertexBuffer** method, which takes pool (memory class) and usage parameters.
CreateVertexBuffer can also be created with a specified flexible vertex format (FVF) code for use in fixed function vertex processing, or as the output of process

vertices. For details, see FVF Vertex Buffers and FVF Usage Settings for Destination Vertex Buffers.

The D3DUSAGE_SOFTWAREPROCESSING flag can be set when mixed-mode or software vertex processing (D3DCREATE_MIXED_VERTEXPROCESSING / D3DCREATE_SOFTWARE_VERTEXPROCESSING) is enabled for that device. D3DUSAGE_SOFTWAREPROCESSING must be set for buffers to be used with software vertex processing in mixed mode, but it should not be set for the best possible performance when using hardware vertex processing in mixed mode. (D3DCREATE_HARDWARE_VERTEXPROCESSING). However, setting D3DUSAGE_SOFTWAREPROCESSING is the only option when a single buffer is to be used with both hardware and software vertex processing. D3DUSAGE_SOFTWAREPROCESSING is allowed for mixed as well as for software devices.
It is possible to force vertex and index buffers into system memory by specifying D3DPOOL_SYSTEMMEM, even when the vertex processing is done in hardware. This is a way to avoid overly large amounts of page-locked memory when a driver is putting these buffers into AGP memory.

FVF Vertex Buffers

[C++]
Setting the FVF parameter of the **IDirect3DDevice8::CreateVertexBuffer** method to a nonzero value, which must be a valid FVF code, indicates that the buffer content is to be characterized by an FVF code. A vertex buffer that is created with an FVF code is referred to as an FVF vertex buffer. Some methods or uses of **IDirect3DDevice8** require FVF vertex buffers, and others require non-FVF vertex buffers. FVF vertex buffers are required as the destination vertex buffer argument for **IDirect3DDevice8::ProcessVertices**.
FVF vertex buffers can be bound to a source data stream for any stream number. However, FVF vertex buffers are not allowed to be used for inputs to programmed vertex shaders
The presence of the D3DFVF_XYZRHW component on FVF vertex buffers indicates that the vertices in that buffer have been processed. Vertex buffers used for **ProcessVertices** destination vertex buffers must be post-processed. Vertex buffers used for fixed function shader inputs can be either pre- or post-processed. If the vertex buffer is post-processed, then the shader is effectively bypassed and the data is passed directly to the primitive clipping and setup module.

[Visual Basic]
Setting the FVF parameter of the **Direct3DDevice8.CreateVertexBuffer** method to a nonzero value, which must be a valid FVF code, indicates that the buffer content is to be characterized by an FVF code. A vertex buffer that is created with an FVF code is referred to as an FVF vertex buffer. Some methods or uses of **Direct3DDevice8** require FVF vertex buffers, and others require non-FVF vertex buffers. FVF vertex

buffers are required as the destination vertex buffer argument for **Direct3DDevice8.ProcessVertices**.
FVF vertex buffers can be bound to a source data stream for any stream number. However, FVF vertex buffers are not allowed to be used for inputs to programmed vertex shaders
The presence of the D3DFVF_XYZRHW component on FVF vertex buffers indicates that the vertices in that buffer have been processed. Vertex buffers used for **ProcessVertices** destination vertex buffers must be post-processed. Vertex buffers used for fixed function shader inputs can be either pre- or post-processed. If the vertex buffer is post-processed, then the shader is effectively bypassed and the data is passed directly to the primitive clipping and setup module.

FVF vertex buffers can be used with vertex shaders. Also, vertex streams can represent the same vertex formats that non-FVF vertex buffers can. They do not have to be used to input data from separate vertex buffers. The additional flexibility of the new vertex streams enables applications that need to keep their data separate to work better, but it is not required. If the application can maintain interleaved data in advance, then that is a performance boost. If the application will only interleave the data before every rendering call, then it should enable the application programming interface (API) or hardware to do this with multiple streams.
The most important things with vertex performance is to use a 32-byte vertex, and to maintain good cache ordering.

Device Types and Vertex Processing Requirements

The performance of vertex processing operations, including transformation and lighting, depends heavily on where the vertex buffer exists in memory and what type of rendering device is being used. Applications control the memory allocation for vertex buffers when they are created. When the D3DPOOL_SYSTEMMEM memory flag is set, the vertex buffer is created in system memory. When the D3DPOOL_DEFAULT memory flag is used, the device driver determines where the memory for the vertex buffer is best allocated, often referred to as driver-optimal memory. Driver-optimal memory can be local video memory, nonlocal video memory, or system memory.

[C++]
Setting the D3DUSAGE_SOFTWAREPROCESSING behavior flag when calling the **IDirect3DDevice8::CreateVertexBuffer** method specifies that the vertex buffer is to be used with software vertex processing. This flag is required for software vertex processing in mixed vertex processing mode. This flag is allowed in software vertex processing mode and disallowed in hardware vertex processing mode. Vertex buffers used with software vertex processing include the following:

- All input streams for **IDirect3DDevice8::ProcessVertices**.
- All input streams for **IDirect3DDevice8::DrawPrimitive** and **IDirect3DDevice8::DrawIndexedPrimitive** when software vertex

processing. For more information, see D3DRS_SOFTWAREVERTEXPROCESSING.

[Visual Basic]
Setting the D3DUSAGE_SOFTWAREPROCESSING behavior flag when calling the **Direct3DDevice8.CreateVertexBuffer** method specifies that the vertex buffer is to be used with software vertex processing. This flag is required for software vertex processing in mixed vertex processing mode. This flag is allowed in software vertex processing mode and disallowed in hardware vertex processing mode. Vertex buffers used with software vertex processing include the following:

- All input streams for **Direct3DDevice8.ProcessVertices**.
- All input streams for **Direct3DDevice8.DrawPrimitive** and **Direct3DDevice8.DrawIndexedPrimitive** when software vertex processing. For more information, see D3DRS_SOFTWAREVERTEXPROCESSING.

The reasoning you use to determine the memory location—system or driver optimal—for vertex buffers is the same as that for textures. Vertex processing, including transformation and lighting, in hardware works best when the vertex buffers are allocated in driver-optimal memory, while software vertex processing works best with vertex buffers allocated in system memory. For textures, hardware rasterization works best when textures are allocated in driver-optimal memory, while software rasterization works best with system-memory textures.

[C++]
Note
Microsoft® Direct3D® for Microsoft DirectX® 8.0 supports standalone processing of vertices, without rendering any primitive with the **ProcessVertices** method. This standalone vertex processing is always performed in software on the host processor. Because of this, vertex buffers used as sources set with **IDirect3DDevice8::SetStreamSource** must be created with the D3DUSAGE_SOFTWAREPROCESSING flag. The functionality provided by **ProcessVertices** is identical to that of the **IDirect3DDevice8::DrawPrimitive** and **IDirect3DDevice8::DrawIndexedPrimitive** methods while using software vertex processing. For more information, see Processing Vertices.

[Visual Basic]

Note
Microsoft® Direct3D® for DirectX® 8.0 supports standalone processing of vertices, without rendering any primitive with the **ProcessVertices** method. This standalone vertex processing is always performed in software on the host processor. Because of this, vertex buffers used as sources set with **Direct3DDevice8.SetStreamSource**

must be created with the D3DUSAGE_SOFTWAREPROCESSING flag. The functionality provided by **ProcessVertices** is identical to that of the **Direct3DDevice8.DrawPrimitive** and **Direct3DDevice8.DrawIndexedPrimitive** methods while using software vertex processing. For more information, see Processing Vertices.

If your application performs its own vertex processing and passes transformed, lit, and clipped vertices to rendering methods, then the application can directly write vertices to a vertex buffer allocated in driver-optimal memory. This technique prevents a redundant copy operation later. Note that this technique will not work well if your application reads data back from a vertex buffer, because read operations done by the host from driver-optimal memory can be very slow. Therefore, if your application needs to read during processing or writes data to the buffer erratically, a system-memory vertex buffer is a better choice.

[C++]
When using the Direct3D vertex processing features (by passing untransformed vertices to vertex-buffer rendering methods), processing can occur in either hardware or software depending on the device type and device creation flags. It is recommended that vertex buffers be allocated in pool D3DPOOL_DEFAULT for best performance in virtually all cases. When a device is using hardware vertex processing, there is a number of additional optimizations that may be done based on the flags D3DUSAGE_DYNAMIC and D3DUSAGE_WRITEONLY. See **IDirect3DDevice8::CreateVertexBuffer** for more information on using these flags.

[Visual Basic]
When using the Direct3D vertex processing features (by passing untransformed vertices to vertex-buffer rendering methods), processing can occur in either hardware or software depending on the device type and device creation flags. It is recommended that vertex buffers be allocated in pool D3DPOOL_DEFAULT for best performance in virtually all cases. When a device is using hardware vertex processing, there is a number of additional optimizations that may be done based on the flags D3DUSAGE_DYNAMIC and D3DUSAGE_WRITEONLY. See **Direct3DDevice8.CreateVertexBuffer** for more information on using these flags.

Using Vertex Buffers

The following topics discuss common tasks that applications perform when working with vertex buffers.

- Creating a Vertex Buffer
- Accessing the Contents of a Vertex Buffer
- Processing Vertices
- Rendering from a Vertex Buffer

- Retrieving Vertex Buffer Descriptions

Creating a Vertex Buffer

[C++]
You create a vertex buffer object by calling the **IDirect3DDevice8::CreateVertexBuffer** method, which accepts five parameters. The first parameter specifies the vertex buffer length, in bytes. Use the **sizeof** operator to determine the size of a vertex format, in bytes. Consider the following custom vertex format.

```
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    FLOAT rhw;
    DWORD color;
    FLOAT tu, tv;   // The texture coordinates.
};

// Custom FVF, which describes the custom vertex structure.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW | D3DFVF_DIFFUSE |
D3DFVF_TEX1)
```

To create a vertex buffer to hold four CUSTOMVERTEX structures, specify [4*sizeof(CUSTOMVERTEX)] for the *Length* parameter.
The second parameter is a set of usage controls. Among other things, its value determines whether the vertex buffer is capable of containing clipping information—in the form of clip flags—for vertices that exist outside the viewing area. To create a vertex buffer that cannot contain clip flags, include the D3DUSAGE_DONOTCLIP flag for the *Usage* parameter. The D3DUSAGE_DONOTCLIP flag is applied only if you also indicate that the vertex buffer will contain transformed vertices—the D3DFVF_XYZRHW flag is included in the *FVF* parameter. The **CreateVertexBuffer** method ignores the D3DUSAGE_DONOTCLIP flag if you indicate that the buffer will contain untransformed vertices (the D3DFVF_XYZ flag). Clipping flags occupy additional memory, making a clipping-capable vertex buffer slightly larger than a vertex buffer incapable of containing clipping flags. Because these resources are allocated when the vertex buffer is created, you must request a clipping-capable vertex buffer ahead of time.
The third parameter, *FVF*, is a combination of Flexible Vertex Format Flags that describe the vertex format of the vertex buffer. If you specify 0 for this parameter, then the vertex buffer is a non-FVF vertex buffer. For more information, see FVF Vertex Buffers. The fourth parameter describes the memory class into which to place the vertex buffer.
The final parameter that **CreateVertexBuffer** accepts is the address of a variable that will be filled with a pointer to the new **IDirect3DVertexBuffer8** interface of the vertex buffer object, if the call succeeds.

Note

You cannot produce clip flags for a vertex buffer that was created without support for them.

The following C++ code example shows what creating a vertex buffer might look like in code.

```
    // For the purposes of this example, the d3dDevice variable is
    // the address of an IDirect3DDevice8 interface exposed by a
    // Direct3DDevice object, g_pVB is a variable of type
    // LPDIRECT3DVERTEXBUFFER8.

// The custom vertex type
struct CUSTOMVERTEX {
        FLOAT x, y, z;
        FLOAT rhw;
        DWORD color;
        FLOAT tu, tv;   // The texture coordinates
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW | D3DFVF_DIFFUSE |
D3DFVF_TEX1)

    // Create a clipping-capable vertex buffer. Allocate enough memory
    // in the  default memory pool to hold three CUSTOMVERTEX
    // structures.
    if( FAILED( d3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
                                        0 /* Usage */, D3DFVF_CUSTOMVERTEX,
                                        D3DPOOL_DEFAULT, &g_pVB ) ) )
        return E_FAIL;
```

[Visual Basic]

You create a vertex buffer object by calling the **Direct3DDevice8.CreateVertexBuffer** method, which accepts five parameters. The first parameter specifies the vertex buffer length, in bytes. Use the **Len** Microsoft Visual Basic® function to determine the size of a vertex format, in bytes. Consider the following custom vertex format.

```
Private Type CUSTOMVERTEX
    x As Single         ' x in screen space
    y As Single         ' y in screen space
    z  As Single        ' normalized z
    rhw As Single       ' normalized z rhw
    color As Long       ' vertex color
    tu As Single        ' texture coordinate
    tv As Single        ' texture coordinate
End Type
```

```
' Custom FVF, which describes the custom vertex structure.
Const D3DFVF_CUSTOMVERTEX = (D3DFVF_XYZRHW Or D3DFVF_DIFFUSE Or
D3DFVF_TEX1)
```

To create a vertex buffer to hold four CUSTOMVERTEX structures, specify [4*sizeof(CUSTOMVERTEX)] for the *Length* parameter.
The second parameter is a set of usage controls. Among other things, its value determines if the vertex buffer is capable of containing clipping information—in the form of clip flags—for vertices that exist outside the viewing area. To create a vertex buffer that cannot contain clip flags, include the D3DUSAGE_DONOTCLIP flag for the *Usage* parameter. The D3DUSAGE_DONOTCLIP flag is applied only if you also indicate that the vertex buffer will contain transformed vertices—the D3DFVF_XYZRHW flag is included in the *FVF* parameter. The **CreateVertexBuffer** method ignores the D3DUSAGE_DONOTCLIP flag if you indicate that the buffer will contain untransformed vertices (the D3DFVF_XYZ flag). Clipping flags occupy additional memory, making a clipping-capable vertex buffer slightly larger than a vertex buffer incapable of containing clipping flags. Because these resources are allocated when the vertex buffer is created, you must request a clipping-capable vertex buffer ahead of time.
The third parameter, *FVF*, is a combination of Flexible Vertex Format Flags that describe the vertex format of the vertex buffer. If you specify 0 for this parameter, then the vertex buffer is a non-FVF vertex buffer. For more information, see FVF Vertex Buffers. The fourth parameter describes the memory class into which to place the vertex buffer.
The final parameter that **CreateVertexBuffer** accepts is the address of a variable to fill with a pointer to the new **Direct3DVertexBuffer8** interface of the vertex buffer object, if the call succeeds.

Note

You cannot produce clip flags for a vertex buffer that was created without support for them.

The following Visual Basic code example shows what creating a vertex buffer might look like in code.

```
'
' For the purposes of this example, the m_d3dDevice variable is
' a valid Direct3DDevice8 object.
'


' The custom vertex type
Private Type CUSTOMVERTEX
    x As Single
    y As Single
    z As Single
    rhw As Single
    color As Long
    tu As Single
    tv As Single
```

```
End Type

Const D3DFVF_CUSTOMVERTEX = (D3DFVF_XYZRHW | D3DFVF_DIFFUSE |
D3DFVF_TEX1)

' Create a clipping-capable vertex buffer. Allocate enough memory
' in the default memory pool to hold three CUSTOMVERTEX structures.
VB = m_d3dDevice.CreateVertexBuffer( 3*len(CUSTOMVERTEX), _
                                0 /* Usage */, D3DFVF_CUSTOMVERTEX, _
                                D3DPOOL_DEFAULT)
```

Accessing the Contents of a Vertex Buffer

[C++]

Vertex buffer objects enable applications to directly access the memory allocated for vertex data. You can retrieve a pointer to vertex buffer memory by calling the **IDirect3DVertexBuffer8::Lock** method, and then accessing the memory as needed to fill the buffer with new vertex data or to read any data it already contains. The **Lock** method accepts four parameters. The first, *OffsetToLock*, is the offset into the vertex data. The second parameter is the size, measured in bytes, of the vertex data. The third parameter accepted by the **Lock** method, *ppbData*, is the address of a **BYTE** pointer that points to the vertex data, if the call succeeds.

The last parameter, *Flags*, tells the system how the memory should be locked. You can use it to indicate how the application will access the data in the buffer. Specify constants for the *Flags* parameter according to the way the vertex data will be accessed. This allows the driver to lock the memory and provide the best performance, given the requested access type. Use D3DLOCK_READONLY flag if your application will read only from the vertex buffer memory. Including this flag enables Microsoft® Direct3D® to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.

After you finish filling or reading the vertex data, call the **IDirect3DVertexBuffer8::Unlock** method, as shown in the following code example.

```
// This code example assumes the g_pVB is a variable of type
// LPDIRECT3DVERTEXBUFFER8 and that g_Vertices has been properly
// initialized with vertices.

// To fill the vertex buffer, you need to lock the buffer to
// gain access to the vertices. This mechanism is required because
// vertex buffers may be in device memory.
VOID* pVertices;

if( FAILED( g_pVB->Lock( 0,              // Fill from the start of
                                 // the buffer.
                         sizeof(g_Vertices), // Size of the data to
```

```
                    // load.
                (BYTE**)&pVertices, // Returned vertex data.
                0 ) ) )         // Send default flags to
                                // the lock.
  return E_FAIL;

memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
g_pVB->Unlock();
```

Performance Notes

If you create a vertex buffer with the D3DUSAGE_WRITEONLY flag, do not use the D3DLOCK_READONLY locking flag. Use the D3DLOCK_READONLY flag if your application will read only from the vertex buffer memory. Including this flag enables Direct3D to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.

See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

In C++, because you directly access the memory allocated for the vertex buffer, make sure your application properly accesses the allocated memory. Otherwise, you risk rendering that memory invalid. Use the stride of the vertex format that your application uses to move from one vertex in the allocated buffer to another. The vertex buffer memory is a simple array of vertices specified in flexible vertex format. Use the stride of whatever vertex format structure you define. You can calculate the stride of each vertex at run time by examining the Flexible Vertex Format Flags contained in the vertex buffer description. The following table shows the size for each vertex component.

Vertex Format Flag	Size
D3DFVF_DIFFUSE	**sizeof**(**DWORD**)
D3DFVF_NORMAL	**sizeof**(**float**) × 3
D3DFVF_SPECULAR	**sizeof**(**DWORD**)
D3DFVF_TEX*n*	**sizeof**(**float**) × *n* × *t*
D3DFVF_XYZ	**sizeof**(**float**) × 3
D3DFVF_XYZRHW	**sizeof**(**float**) × 4

The number of texture coordinates present in the vertex format is described by the D3DFVF_TEX*n* flags (where *n* is a value from 0 to 8). Multiply the number of texture coordinate sets by the size of one set of texture coordinates, which can range from one to four floats, to calculate the memory required for that number of texture coordinates.

Use the total vertex stride to increment and decrement the memory pointer as needed to access particular vertices.

[Visual Basic]
Vertex buffers enable Microsoft® Visual Basic® applications to easily update the vertex data they contain. To lock, fill, and unlock vertex buffers use the **D3DVertexBuffer8SetData** helper function. **D3DVertexBuffer8SetData** accepts five parameters, The first, *Vbuffer*, is the **Direct3DVertexBuffer8** object that contains the vertex data. The second parameter, *OffsetToLock*, is the offset into the vertex data. The third parameter is the size, measured in bytes, of the vertex data. The fourth parameter, *Flags*, tells the system how the memory should be locked. You can use it to indicate how the application will access the data in the buffer. Specify constants for the *Flags* parameter according to the way the vertex data will be accessed by your application. This allows the driver to lock the memory and provide the best performance given the requested access type. Use D3DLOCK_READONLY flag if your application will read only from the vertex buffer memory. Including this flag enables Microsoft® Direct3D® to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.
The last parameter accepted by the **D3DVertexBuffer8SetData** helper function, *Data*, is the first element of an array of data to be loaded into the vertex buffer. This parameter is of type Any. To use it properly you must specify the first element of the array for *Data*, as shown in the following code example.

```
Dim Vertices(3) As CUSTOMVERTEX
Dim VertexSizeInBytes As Long

' Determine the length of the vertex data.
VertexSizeInBytes = Len(Vertices(0))

' Fill the vertices with data.
    .
    .
    .

' This code example assumes that g_VB is a variable containing a
' valid reference to a Direct3DVertexBuffer8 object.

' VBuffer=g_VB          The vertex buffer to fill with data.
' Offset=0              Start filling from the start of the buffer.
' Size=VertSizeInBytes*3   Copy three CUSTOMVERTEX types to the buffer.
' Flags=0               Send default flags to the lock operation.
' data=Vertices(0)      This parameter is of type Any. In order
'                       to use it, send the first element in
'                       the array.

D3DVertexBuffer8SetData g_VB, 0, VertexSizeInBytes * 3, 0, Vertices(0)
```

The vertex buffer memory is a simple array of vertices specified in flexible vertex format. The number of texture coordinates present in the vertex format is described by the D3DFVF_TEX*n* flags, where *n* is a value from 0 to 8. Multiply the number of

texture coordinate sets by the size of one set of texture coordinates, which can range from one to four floats, to calculate the memory required for that number of texture coordinates.

Performance Notes

If you create a vertex buffer with the D3DUSAGE_WRITEONLY flag, do not use the D3DLOCK_READONLY locking flag. Use the D3DLOCK_READONLY flag if your application will read only from the vertex buffer memory. Including this flag enables Direct3D to optimize its internal procedures, given that access to the memory will be read-only.
See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

To lock, read, and unlock the contents of vertex buffers, use the **D3DVertexBuffer8GetData** helper function.

Processing Vertices

Processing the vertices in a vertex buffer applies the current transformation matrices for the device. Vertex operations such as lighting, generating clip flags, and updating extents can also be applied, optionally. Information is divided into the following topics.

- Fixed Function Processing Vertices
- Programmable Processing Vertices

Fixed Function Processing Vertices

This section discusses fixed function vertex processing. The following topics are discussed.

- About Fixed Function Processing Vertices
- FVF Usage Settings for Destination Vertex Buffers

About Fixed Function Processing Vertices

[C++]
In the fixed function vertex pipeline, processing the vertices in a vertex buffer applies the current transformation matrices for the device. Vertex operations such as lighting, generating clip flags, and updating extents can also be applied, optionally. When using fixed function vertex processing, modifying the elements in the destination vertex buffer is controlled by the D3DPV_DONOTCOPYDATA flag. This flag applies only to fixed function vertex processing. The **IDirect3DDevice8** interface exposes the **IDirect3DDevice8::ProcessVertices** method to process vertices. You process vertices from a vertex shader to the set of input data streams, generating a single stream of interleaved vertex data to the destination vertex buffer by calling the **ProcessVertices** method. The method accepts five parameters that describe the location and quantity of vertices that the method targets, the destination vertex buffer,

and the processing options. After the call, the destination buffer contains the processed vertex data.
The first, second, and third parameters, *SrcStartIndex, DestIndex,* and *VertexCount*, reflect the index of the first vertex to load, the index within the destination buffer at which the vertices will be placed, and the total number of vertices to process and place in the destination buffer. The fourth parameter, *pDestBuffer,* should be set to the address of the **IDirect3DVertexBuffer8** of the vertex buffer object that will receive the source vertices. The *SrcStartIndex* specifies the index at which the method should start processing vertices.
The final parameter, *Flags*, determines special processing options for the method. You can set this parameter to 0 for default vertex processing, or to D3DPV_DONOTCOPYDATA to optimize processing in some situations. When you set *Flags* to 0, vertex components of the destination vertex buffer's vertex format that are not affected by the vertex operation are still copied from the vertex shader or set to 0. However, when using D3DPV_DONOTCOPYDATA, **ProcessVertices** does not overwrite color and texture coordinate information in the destination buffer unless this data is generated by Microsoft® Direct3D®. Diffuse color is generated when lighting is enabled, that is, D3DRS_LIGHTING is set to TRUE. Specular color is generated when lighting is enabled and specular is enabled, that is, D3DRS_SPECULARENABLE and D3DRS_LIGHTING are set to TRUE. Specular color is also generated when fog is enabled. Texture coordinates are generated when texture transform or texture generation is enabled. **ProcessVertices** uses the current render states to determine what vertex processing should be done.
To determine the vertex processing limitations of a device, query the **VertexProcessingCaps** member of the **D3DCAPS8** structure for its supported vertex processing capabilities.

[Visual Basic]
Processing the vertices in a vertex buffer applies the current transformation matrices for the device, and it can optionally apply vertex operations such as lighting, generating clip flags, and updating extents. When using fixed function vertex processing, modifying the elements in the destination vertex buffer is controlled by the D3DPV_DONOTCOPYDATA flag. This flag applies only to fixed function vertex processing. The **Direct3DDevice8** class exposes the **Direct3DDevice8.ProcessVertices** method to process vertices. You process vertices from a vertex shader to the set of input data streams, generating a single stream of interleaved vertex data to the destination vertex buffer by calling the **ProcessVertices** method. The method accepts five parameters that describe the location and quantity of vertices that the method targets, the destination vertex buffer, and the processing options. After the call, the destination buffer contains the processed vertex data.
The first, second, and third parameters, *SrcStartIndex, DestIndex,* and *VertexCount*, reflect the index of the first vertex to load, the index in the destination buffer at which the vertices will be placed, and the total number of vertices to process and place in the destination buffer. The fourth parameter, *DestBuffer,* should be set to the **Direct3DVertexBuffer8** of the vertex buffer object that will receive the source vertices. The *SrcStartIndex* specifies the index at which the method should start processing vertices.

The final parameter, *Flags*, determines special processing options for the method. You can set this parameter to 0 for default vertex processing, or to D3DPV_DONOTCOPYDATA to optimize processing in some situations. When you set *Flags* to 0, vertex components of the destination vertex buffer's vertex format that are not affected by the vertex operation are still copied from the vertex shader or set to 0. However, when using D3DPV_DONOTCOPYDATA, **ProcessVertices** does not overwrite color and texture coordinate information in the destination buffer unless this data is generated by Microsoft® Direct3D®. Diffuse color is generated when lighting is enabled, that is, D3DRS_LIGHTING is set to TRUE. Specular color is generated when lighting is enabled and specular is enabled, that is, D3DRS_SPECULARENABLE and D3DRS_LIGHTING are set to TRUE. Specular color is also generated when fog is enabled. Texture coordinates are generated when texture transform or texture generation is enabled. **ProcessVertices** uses the current render states to determine what vertex processing should be done.
To determine the vertex processing limitations of a device, query the **VertexProcessingCaps** member of the **D3DCAPS8** structure for its supported vertex processing capabilities.

FVF Usage Settings for Destination Vertex Buffers

[C++]
The **IDirect3DDevice8::ProcessVertices** method requires specific settings for the Flexible Vertex Format Flags of the destination vertex buffer. The flexible vertex format (FVF) usage settings must be compatible with the current settings for vertex processing.
For fixed function vertex processing, **ProcessVertices** requires the following FVF settings.

- Position type is always D3DFVF_XYZRHW; so, D3DFVF_XYZ and D3DFVF_XYZB1 through D3DFVF_XYZB5 are invalid.
- The D3DFVF_NORMAL, D3DFVF_RESERVED0, and D3DFVF_RESERVED2 flags must not be set.
- The D3DFVF_DIFFUSE flag must be set if an **OR** operation of the following conditions returns true.
  - Lighting is enabled, that is, D3DRS_LIGHTING is true.
  - Lighting is disabled, and diffuse color is present in the input vertex streams, and D3DPV_DONOTCOPYDATA is not set.
- The D3DFVF_SPECULAR flag must be set if an **OR** operation of the following conditions returns true.
  - Lighting is enabled and specular color is enabled, that is, D3DRS_SPECULARENABLE is true.
  - Lighting is disabled, and specular color is present in the input vertex streams, and D3DPV_DONOTCOPYDATA is not set.
  - Vertex fog is enabled, that is, D3DRS_FOGVERTEXMODE is not set to D3DFOG_NONE.

In addition, the texture coordinate count must be set in the following manner.

- If texture transform and texture generation are disabled for all active texture stages, and the D3DPV_DONOTCOPYDATA is not set, then the number and type of output texture coordinates are required to match those of the input vertex texture coordinates. If D3DPV_DONOTCOPYDATA is set and texture transform and texture generation are disabled, then the output texture coordinates are ignored.
- If texture transform or texture generation is enabled for any active texture stages, the output vertex might need to contain more texture coordinate sets than the input vertex. This is due to the proliferation of texture coordinates from those being generated by texture generation or derived by texture transforms. Note that a similar proliferation of texture coordinates occurs during **DrawPrimitive** calls, but is not visible to the application programmer. In this case, Microsoft® Direct3D® generates a new set of texture coordinates. The new set of texture coordinates is derived by stepping through the texture stages and analyzing the settings for texture generation, texture transformation, and texture coordinate index to determine if a unique set of texture coordinates is required for that stage. Each time a new set is required it is allocated in increasing order. Note that the maximum, and typical, requirement is one set per stage, although it might be less due to sharing of nontransformed texture coordinates through D3DTSS_TEXCOORDINDEX.

Thus, for each texture stage, a new set of texture coordinates is generated if a texture is bound to that stage and any of the following conditions are true.

- Texture generation is enabled for that stage.
- Texture transformation is enabled for that stage.
- Nontransformed input texture coordinates are referenced through D3DTSS_TEXCOORDINDEX for the first time.

When Direct3D is generating texture coordinates, the application is required to perform the following actions.

1. Use a destination vertex buffer with the appropriate FVF usage.
2. Reprogram the D3DTSS_TEXCOORDINDEX of the texture stage according to the placement of the post -processed texture coordinates. Note that the reprogramming of the D3DTSS_TEXCOORDINDEX setting occurs when the processed vertex buffer is used in subsequent **IDirect3DDevice8::DrawPrimitive** and **IDirect3DDevice8::DrawIndexedPrimitive** calls.

[Visual Basic]
The **Direct3DDevice8.ProcessVertices** method requires specific settings for the Flexible Vertex Format Flags of the destination vertex buffer. The flexible vertex

format (FVF) usage settings must be compatible with the current settings for vertex processing.
For fixed function vertex processing, **ProcessVertices** requires the following FVF settings.

- Position type is always D3DFVF_XYZRHW; so, D3DFVF_XYZ and D3DFVF_XYZB1 through D3DFVF_XYZB5 are invalid.
- The D3DFVF_NORMAL, D3DFVF_RESERVED0, and D3DFVF_RESERVED2 flags must not be set.
- The D3DFVF_DIFFUSE flag must be set if an **OR** operation of the following conditions returns true.
  - Lighting is enabled, that is, D3DRS_LIGHTING is true.
  - Lighting is disabled, and diffuse color is present in the input vertex streams, and D3DPV_DONOTCOPYDATA is not set.
- The D3DFVF_SPECULAR flag must be set if an **OR** of the following conditions returns true.
  - Lighting is enabled and specular color is enabled, that is, D3DRS_SPECULARENABLE is true.
  - Lighting is disabled, and specular color is present in the input vertex streams, and D3DPV_DONOTCOPYDATA is not set.
  - Vertex fog is enabled, that is, D3DRS_FOGVERTEXMODE is not set to D3DFOG_NONE.

In addition, the texture coordinate count must be set in the following manner.

- If texture transform and texture generation are disabled for all active texture stages, and the D3DPV_DONOTCOPYDATA is not set, then the number and type of output texture coordinates are required to match those of the input vertex texture coordinates. If D3DPV_DONOTCOPYDATA is set and texture transform and texture generation are disabled, then the output texture coordinates are ignored.
- If texture transform or texture generation is enabled for any active texture stages, the output vertex might need to contain more texture coordinate sets than the input vertex. This is due to the proliferation of texture coordinates from those being generated by texture generation or derived by texture transforms. Note that a similar proliferation of texture coordinates occurs during **DrawPrimitive** calls, but is not visible to the application programmer. In this case, Microsoft® Direct3D® generates a new set of texture coordinates. The new set of texture coordinates is derived by stepping through the texture stages and analyzing the settings for texture generation, texture transformation, and texture coordinate index to determine if a unique set of texture coordinates is required for that stage. Each time a new set is required, it is allocated in increasing order. Note that the maximum, and typical, requirement is one set per stage, although it might be less due to sharing of nontransformed texture coordinates through D3DTSS_TEXCOORDINDEX.

Thus, for each texture stage, a new set of texture coordinates is generated if a texture is bound to that stage and any of the following conditions are true.

- Texture generation is enabled for that stage.
- Texture transformation is enabled for that stage.
- Nontransformed input texture coordinates are referenced through D3DTSS_TEXCOORDINDEX for the first time.

When Direct3D is generating texture coordinates, the application is required to perform the following actions.

1. Use a destination vertex buffer with the appropriate FVF usage.
2. Reprogram the D3DTSS_TEXCOORDINDEX of the texture stage according to the placement of the post -processed texture coordinates. Note that the reprogramming of the D3DTSS_TEXCOORDINDEX setting occurs when the processed vertex buffer is used in subsequent **Direct3DDevice8.DrawPrimitive** and **Direct3DDevice8.DrawIndexedPrimitive** calls.

Finally, texture coordinate dimensionality (D3DFVF_TEX0 through D3DFVF_TEX8) must be set in the following manner.

- For each texture coordinate set, if texture transform and texture generation are disabled, then the output texture coordinate dimensionality must match the input. If the texture transform is enabled, then the output dimensionality must match the count defined by the D3DTTFF_COUNT1, D3DTTFF_COUNT2, D3DTTFF_COUNT3, or D3DTTFF_COUNT4 settings. If the texture transform is disabled and texture generation is enabled, then the output dimensionality must match the settings for the texture generation mode; currently all modes generate 3 float values.

When **ProcessVertices** fails due to an incompatible destination vertex buffer FVF code, the expected code is printed to the debug output (debug builds only).

Programmable Processing Vertices

[C++]
When using a programmed vertex shader, the elements updated in the destination vertex buffer are controlled by the shader function program. When the application writes to one of the destination vertex registers, the corresponding element within each vertex of the destination vertex buffer is updated. Elements in the destination vertex buffer that are not written to by the application are not modified. **IDirect3DDevice8::ProcessVertices** will fail if the application writes to elements that are not present in the destination vertex buffer.

[Visual Basic]

When using a programmed vertex shader, the elements updated in the destination vertex buffer are controlled by the shader function program. When the application writes to one of the destination vertex registers, the corresponding element within each vertex of the destination vertex buffer is updated. Elements in the destination vertex buffer that are not written to by the application are not modified. **Direct3DDevice8.ProcessVertices** will fail if the application writes to elements that are not present in the destination vertex buffer.

Rendering from a Vertex Buffer

[C++]
Rendering vertex data from a vertex buffer requires a few steps. First, you need to set the stream source by calling the **IDirect3DDevice8::SetStreamSource** method, as shown in the following example.

```
d3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
```

The first parameter of **SetStreamSource** tells Microsoft® Direct3D® the source of the device data stream. The second parameter is the vertex buffer to bind to the data stream. The third parameter is the size of the component, in bytes. In the sample code above, the size of a CUSTOMVERTEX is used for the size of the component.
The next step is to inform Direct3D which vertex shader to use by calling the **IDirect3DDevice8::SetVertexShader** method. The following sample code sets an FVF code for the vertex shader. This informs Direct3D of the types of vertices it is dealing with.

```
d3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
```

After setting the stream source and vertex shader, any draw methods will use the vertex buffer. The code example below shows how to render vertices from a vertex buffer with the **IDirect3DDevice8::DrawPrimitive** method.

```
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
```

The second parameter that **DrawPrimitive** accepts is the index of the first vector in the vertex buffer to load.

[Visual Basic]
Rendering vertex data from a vertex buffer requires a few steps. First, you need to set the stream source by calling the **Direct3DDevice8.SetStreamSource** method, as shown in the following code example.

```
Call m_D3Device.SetStreamSource(0, VB, len(CUSTOMVERTEX))
```

The first parameter of **SetStreamSource** tells Direct3D the source of the device data stream. The second parameter is the vertex buffer to bind to the data stream. The third parameter is the size of the component, in bytes. In the sample code above, the size of a CUSTOMVERTEX is used for the size of the component.

The next step is to inform Direct3D which vertex shader to use by calling the **Direct3DDevice8.SetVertexShader** method. The following sample code sets an FVF code for the vertex shader. This informs Direct3D of the types of vertices it is dealing with.

```
Call m_D3DDevice.SetVertexShader(D3DFVF_CUSTOMVERTEX)
```

After setting the stream source and vertex shader, any draw methods will use the vertex buffer. The code example below shows how to render vertices from a vertex buffer with the **Direct3DDevice8.DrawPrimitive** method.

```
Call m_D3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1)
```

The second parameter that **DrawPrimitive** accepts is the index of the first vector in the vertex buffer to load.

Retrieving Vertex Buffer Descriptions

[C++]
You can retrieve information about a vertex buffer by calling the **IDirect3DVertexBuffer8::GetDesc** method. This method fills the members of the **D3DVERTEXBUFFER_DESC** structure with information about the vertex buffer.

[Visual Basic]
Retrieve information about a vertex buffer by calling the **Direct3DVertexBuffer8.GetDesc** method. This method fills the members of the **D3DVERTEXBUFFER_DESC** structure with information about the vertex buffer.

Index Buffers

This section introduces the concepts necessary to understand and use index buffers in a Microsoft® Direct3D® application. Information is divided into the following sections.

- What Are Index Buffers?
- Index Buffer Descriptions
- Pool and Usage for Index Buffers
- Device Types and Index Processing Requirements
- Using Index Buffers

What Are Index Buffers?

[C++]

Index buffers, represented by the **IDirect3DIndexBuffer8** interface, are memory buffers that contain index data. Index data, or indices, are integer offsets into vertex buffers and are used to render primitives using the **IDirect3DDevice8::DrawIndexedPrimitive** method.

[Visual Basic]
Index buffers, represented by the **Direct3DIndexBuffer8** class, are memory buffers that contain index data. Index data, or indices, are integer offsets into vertex buffers and are used for rendering primitives using the **Direct3DDevice8.DrawIndexedPrimitive** method.

Index Buffer Descriptions

[C++]
An index buffer is described in terms of its capabilities: if it can exist only in system memory, if it is only used for write operations, and the type and number of indices it can contain. These traits are held in a **D3DINDEXBUFFER_DESC** structure.

[Visual Basic]
An index buffer is described in terms of its capabilities: if it can exist only in system memory, if it is only used for write operations, and the type and number of vertices it can contain. These traits are held in a **D3DINDEXBUFFER_DESC** type.

Index buffer descriptions tell your application how an existing buffer was created. You provide an empty description structure for the system to fill with the capabilities of a previously created index buffer. For more information about this task, see Retrieving Index Buffer Descriptions.

[C++]
The **Format** member describes the surface format of the index buffer data.
The **Type** identifies the resource type of the index buffer.
The **Usage** structure member contains general capability flags. The D3DUSAGE_SOFTWAREPROCESSING flag indicates that the index buffer is to be used with software vertex processing. The presence of the D3DUSAGE_WRITEONLY flag in **Usage** indicates that the index buffer memory is used only for write operations. This frees the driver to place the index data in the best memory location to enable fast processing and rendering. If the D3DUSAGE_WRITEONLY flag is not used, the driver is less likely to put the data in a location that is inefficient for read operations. This sacrifices some processing and rendering speed. If this flag is not specified, it is assumed that applications perform read and write operations on the data in the index buffer.

Pool specifies the memory class allocated for the index buffer. The D3DPOOL_SYSTEMMEM flag indicates that the system created the index buffer in system memory.
The **Size** member simply stores the size, in bytes, of the vertex buffer data.

[Visual Basic]
The **Format** member describes the surface format of the index buffer data.
The **Type** identifies the resource type of the index buffer.
The **Usage** structure member contains general capability flags. The D3DUSAGE_SOFTWAREPROCESSING flag indicates that the index buffer is to be used with software vertex processing. The presence of the D3DUSAGE_WRITEONLY flag in **Usage** indicates that the index buffer memory is used only for write operations. This frees the driver to place the index data in the best memory location to enable fast processing and rendering. If the D3DUSAGE_WRITEONLY flag is not used, the driver is less likely to put the data in a location that is inefficient for read operations. This sacrifices some processing and rendering speed. If this flag is not specified, it is assumed that applications perform read and write operations on the data within the index buffer.
Pool specifies the memory class allocated for the index buffer. The D3DPOOL_SYSTEMMEM flag indicates that the system created the index buffer in system memory.
The **Size** member simply stores the size, in bytes, of the index buffer data.

Pool and Usage for Index Buffers

[C++]
You can create index buffers with the **IDirect3DDevice8::CreateIndexBuffer** method, which takes pool (memory class) and usage parameters.

[Visual Basic]
You can create index buffers with the **Direct3DDevice8.CreateIndexBuffer** method, which takes pool (memory class) and usage parameters.

The D3DUSAGE_SOFTWAREPROCESSING flag can be set when mixed-mode or software vertex processing (D3DCREATE_MIXED_VERTEXPROCESSING / D3DCREATE_SOFTWARE_VERTEXPROCESSING) is enabled for that device. D3DUSAGE_SOFTWAREPROCESSING must be set for buffers to be used with software vertex processing in mixed mode, but it should not be set for the best possible performance when using hardware index processing in mixed mode (D3DCREATE_HARDWARE_VERTEXPROCESSING). However, setting D3DUSAGE_SOFTWAREPROCESSING is the only option when a single buffer is used with both hardware and software vertex processing.

D3DUSAGE_SOFTWAREPROCESSING is allowed for mixed and software devices.
It is possible to force vertex and index buffers into system memory by specifying D3DPOOL_SYSTEMMEM, even when the index processing is being done in hardware. This is a way to avoid overly large amounts of page-locked memory when a driver is putting these buffers into AGP memory.

Device Types and Index Processing Requirements

The performance of index processing operations depends heavily on where the index buffer exists in memory and what type of rendering device is being used. Applications control the memory allocation for index buffers when they are created. When the D3DPOOL_SYSTEMMEM memory flag is set, the index buffer is created in system memory. When the D3DPOOL_DEFAULT memory flag is used, the device driver determines where the memory for the index buffer is best allocated, often referred to as driver-optimal memory. Driver-optimal memory can be local video memory, nonlocal video memory, or system memory.

[C++]
Setting the D3DUSAGE_SOFTWAREPROCESSING behavior flag when calling the **IDirect3DDevice8::CreateIndexBuffer** method specifies that the index buffer is to be used with software vertex processing. This flag is required in mixed mode vertex processing (D3DCREATE_MIXED_VERTEXPROCESSING) when software vertex processing is used.

[Visual Basic]
Setting the D3DUSAGE_SOFTWAREPROCESSING behavior flag when calling the **Direct3DDevice8.CreateIndexBuffer** method specifies that the vertex buffer is to be used with software vertex processing. This flag is required in mixed mode vertex processing (D3DCREATE_MIXED_VERTEXPROCESSING) when software vertex processing is used.

The application can directly write indices to a index buffer allocated in driver-optimal memory. This technique prevents a redundant copy operation later. This technique does not work well if your application reads data back from an index buffer, because read operations done by the host from driver-optimal memory can be very slow. Therefore, if your application needs to read during processing or writes data to the buffer erratically, a system-memory index buffer is a better choice.

Note
Always use D3DPOOL_DEFAULT, except when you don't want to expend video memory or expand large amounts of page-locked RAM when the driver is putting vertex or index buffers into AGP memory.

Using Index Buffers

The following topics discuss common tasks that an application performs when working with vertex buffers.

- Creating an Index Buffer
- Accessing the Contents of an Index Buffer
- Rendering from an Index Buffer
- Retrieving Index Buffer Descriptions

Creating an Index Buffer

[C++]
Create an index buffer object by calling the **IDirect3DDevice8::CreateIndexBuffer** method, which accepts five parameters. The first parameter specifies the index buffer length, in bytes.
The second parameter is a set of usage controls. Among other things, its value determines whether the vertices being referred to by the indices are capable of containing clipping information. To improve performance, specify D3DUSAGE_DONOTCLIP when clipping is not required.
The third parameter is either the D3DFMT_INDEX16 or D3DFMT_INDEX32 member of the **D3DFORMAT** enumerated type that specifies the size of each index.
The fourth parameter is a member of the **D3DPOOL** enumerated type that tells the system where in memory to place the new index buffer.
The final parameter that **CreateIndexBuffer** accepts is the address of a variable that is filled with a pointer to the new **IDirect3DIndexBuffer8** interface of the vertex buffer object, if the call succeeds.
The following C++ code example shows what creating a index buffer might look like in code.

```
/*
 * For the purposes of this example, the d3dDevice variable is the
 * address of an IDirect3DDevice8 interface exposed by a
 * Direct3DDevice object, g_IB is a variable of type
 * LPDIRECT3DINDEXBUFFER8.
 */

if( FAILED( d3dDevice->CreateIndexBuffer( 16384 *sizeof(WORD),
                                          D3DUSAGE_WRITEONLY, D3DFMT_INDEX16,
                                          D3DPOOL_DEFAULT, &g_IB ) ) )
    return E_FAIL;
```

[Visual Basic]
Create an index buffer object by calling the **Direct3DDevice8.CreateIndexBuffer** method, which accepts five parameters. The first parameter specifies the index buffer length, in bytes.

The second parameter is a set of usage controls. Among other things, its value determines whether the vertices being referred to by the indices are capable of containing clipping information. To improve performance, specify D3DUSAGE_DONOTCLIP when clipping is not required.
The third parameter is either the D3DFMT_INDEX16 or D3DFMT_INDEX32 member of the **CONST_D3DFORMAT** enumerated type that specifies the size of each index.
The fourth parameter is a member of the **CONST_D3DPOOL** enumerated type that tells the system where in memory to place the new index buffer.
The following Visual Basic code example shows what creating a index buffer might look like in code.

```
'
' For the purposes of this example, the m_D3DDevice variable is
' an IDirect3DDevice8 interface exposed by a Direct3DDevice
' object. IB is a variable of type Direct3DIndexBuffer8.
'

Set IB = m_D3DDevice.CreateIndexBuffer(16834, D3DUSAGE_WRITEONLY,
                                       D3DFMT_INDEX16, D3DPOOL_DEFAULT)
```

Accessing the Contents of an Index Buffer

[C++]
Index buffer objects enable applications to directly access the memory allocated for index data. You can retrieve a pointer to index buffer memory by calling the **IDirect3DIndexBuffer8::Lock** method, and then accessing the memory as needed to fill the buffer with new index data or to read any data it contains. The **Lock** method accepts four parameters. The first, *OffsetToLock*, is the offset into the index data. The second parameter is the size, measured in bytes, of the index data. The third parameter accepted by the **Lock** method, *ppbData*, is the address of a **BYTE** pointer filled with a pointer to the index data, if the call succeeds.
The last parameter, *Flags*, tells the system how the memory should be locked. You can use it to indicate how the application accesses the data in the buffer. Specify constants for the *Flags* parameter according to the way the index data will be accessed by your application. This allows the driver to lock the memory and provide the best performance given the requested access type. Use D3DLOCK_READONLY flag if your application will read only from the index buffer memory. Including this flag enables Microsoft® Direct3D® to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.
After you fill or read the index data, call the **IDirect3DIndexBuffer8::Unlock** method, as shown in the following code example.

```
// This code example assumes the IB is a variable of type
// LPDIRECT3DINDEXBUFFER8 and that g_Indices has been properly
// initialized with indices.
```

```
// To fill the index buffer, you must lock the buffer to gain
// access to the indices. This mechanism is required because index
// buffers may be in device memory.

VOID* pIndices;

if( FAILED( IB->Lock( 0,              // Fill from start of the buffer.
                sizeof(g_Indices), // Size of the data to load.
                (BYTE**)&pIndices, // Returned index data.
                0 ) ) )          // Send default flags to the lock.
  return E_FAIL;

memcpy( pIndices, g_Indices, sizeof(g_Vertices) );
IB->Unlock();
```

Note

If you create an index buffer with the D3DUSAGE_WRITEONLY flag, do not use the D3DLOCK_READONLY locking flag. Use the D3DLOCK_READONLY flag if your application will read only from the index buffer memory. Including this flag enables Direct3D to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.

See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

In C++, because you directly access the memory allocated for the index buffer, make sure your application properly accesses the allocated memory. Otherwise, you risk rendering that memory invalid. Use the stride of the index format your application uses to move from one index in the allocated buffer to another.

[Visual Basic]

Index buffers enable Microsoft® Visual Basic® applications to easily update the index data they contain. To lock, fill, and unlock index buffers use the **D3DIndexBuffer8SetData** helper function. **D3DIndexBuffer8SetData** accepts five parameters, The first, *Ibuffer*, is the **Direct3DIndexBuffer8** object that contains the index data. The second parameter, *Offset*, is the offset into the index data. The third parameter is the size, measured in bytes, of the index data.

The fourth parameter, *Flags*, tells the system how the memory should be locked. You can use it to indicate how the application accesses the data in the buffer. Specify constants for the *Flags* parameter according to the way the index data will be accessed by your application. This allows the driver to lock the memory and provide the best performance given the requested access type. Use D3DLOCK_READONLY flag if your application will read only from the index buffer memory. Including this

flag enables Microsoft® Direct3D® to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.
The last parameter accepted by the **D3DIndexBuffer8SetData** helper function, *Data*, is the first element of an array of data to load into the index buffer. Note this parameter is of type Any. To use it properly, you must specify the first element of the array for *Data*, as shown in the following code example.

```
Dim Indices(2) As CUSTOMINDEX
Dim IndexSizeInBytes As Long

'Determine the length of the index data.
IndexSizeInBytes = Len(Indices(0))

'Copy indices into the index buffer.
D3DIndexBuffer8SetData VB, 0, IndexSizeInBytes * 3, 0, Indices(0)
```

Note

If you create an index buffer with the D3DUSAGE_WRITEONLY flag, do not use the D3DLOCK_READONLY locking flag. Use the D3DLOCK_READONLY flag if your application will read only from the index buffer memory. Including this flag enables Direct3D to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only.
See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

To lock, read, and unlock the contents of index buffers, use the **D3DIndexBuffer8GetData** helper function.

Rendering from an Index Buffer

[C++]
Rendering index data from an index buffer requires a few steps. First, you need to set the stream source by calling the **IDirect3DDevice8::SetStreamSource** method.

```
d3dDevice->SetStreamSource( 0, VB, sizeof(CUSTOMVERTEX) );
```

The first parameter of **SetStreamSource** tells Microsoft® Direct3D® the source of the device data stream. The second parameter is the vertex buffer to bind to the data stream. The third parameter is the size of the component, in bytes. In the sample code above, the size of a CUSTOMVERTEX is used for the size of the component.
The next step is to call the **IDirect3DDevice8::SetIndices** method to set the source of the index data.

```
d3dDevice->SetIndices( IB, 0 );
```

The first parameter that **SetIndices** accepts is the address of the index buffer to set. The second parameter is the starting point in the vertex stream.

After setting the stream and indices source, use the **IDirect3DDevice8::DrawIndexedPrimitive** method to render vertices that use indices from the index buffer.

```
d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0,
                        dwVertices, 0, dwIndices / 3);
```

The second parameter that **DrawPrimitive** accepts is the minimum vertex index for vertices used during this call. The third parameter is the number of indices to use during this call starting from *BaseVertexIndex* + *MinIndex*. The fourth parameter is the location in the index array to start reading indices. The final parameter that **DrawPrimitive** accepts is the number of primitives to render. This parameter is a function of the primitive count and the primitive type. The code sample above uses triangles, so the number of primitives to render is the number of indices divided by three.

[Visual Basic]

Rendering index data from an index buffer requires a few steps. First, you must set the stream source by calling the **Direct3DDevice8.SetStreamSource** method.

```
Call m_D3DDevice.SetStreamSource( 0, VB, len(CUSTOMVERTEX) )
```

The first parameter of **SetStreamSource** tells Direct3D the source of the device data stream. The second parameter is the vertex buffer to bind to the data stream. The third parameter is the size of the component, in bytes. In the sample code above, the size of a CUSTOMVERTEX is used for the size of the component.

The next step is to call the **Direct3DDevice8.SetIndices** method to set the source of the index data.

```
Call m_D3DDevice.SetIndices( IB, 0 )
```

The first parameter that **SetIndices** accepts is the address of the index buffer to set. The second parameter is the starting point in the vertex stream.

After setting the stream and index sources, use the **Direct3DDevice8.DrawIndexedPrimitive** method to render vertices that use indices from the index buffer.

```
Call m_D3DDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, _
                        0, dwVertices, 0, dwIndices / 3)
```

The second parameter that **DrawPrimitive** accepts is the minimum vertex index for vertices used during this call. The third parameter is the number of indices to use during this call starting from *BaseVertexIndex* + *MinIndex*. The fourth parameter is the location in the index buffer to start reading indices. The final parameter that **DrawPrimitive** accepts is the number of primitives to render. This parameter is a function of the primitive count and the primitive type. The code sample above uses triangles, so the number of primitives to render is the number of indices divided by three.

Retrieving Index Buffer Descriptions

[C++]
Retrieve information about an index buffer by calling the **IDirect3DIndexBuffer8::GetDesc** method. This method fills the members of the **D3DINDEXBUFFER_DESC** structure with information about the vertex buffer.

[Visual Basic]
Retrieve information about an index buffer by calling the **Direct3DIndexBuffer8.GetDesc** method. This method fills the members of the **D3DINDEXBUFFER_DESC** structure with information about the vertex buffer.

Techniques and Special Effects

Microsoft® Direct3D® provides a powerful set of tools that you can use to increase the realistic appearance of a 3-D scene. This section presents information on common special effects that can be produced with Direct3D, but the range of possible effects is not limited to those presented here. The discussion in this section is organized into the following topics.

- Alpha Blending
- Antialiasing
- Bump Mapping
- Environment Mapping
- Fog
- Geometry Blending
- Indexed Vertex Blending
- Stencil Buffer Techniques
- Vertex Tweening
- Colored Lights

Alpha Blending

Alpha blending is used to display an image that has transparent or semi-transparent pixels. In addition to a red, green, and blue color channel, each pixel in an alpha bitmap has a transparency component known as its *alpha channel*. The alpha channel typically contains as many bits as a color channel. For example, an 8-bit alpha channel can represent 256 levels of transparency, from 0 (the entire pixel is transparent) to 255 (the entire pixel is opaque). The list below shows some special effects you can create using alpha blending.

- Billboarding

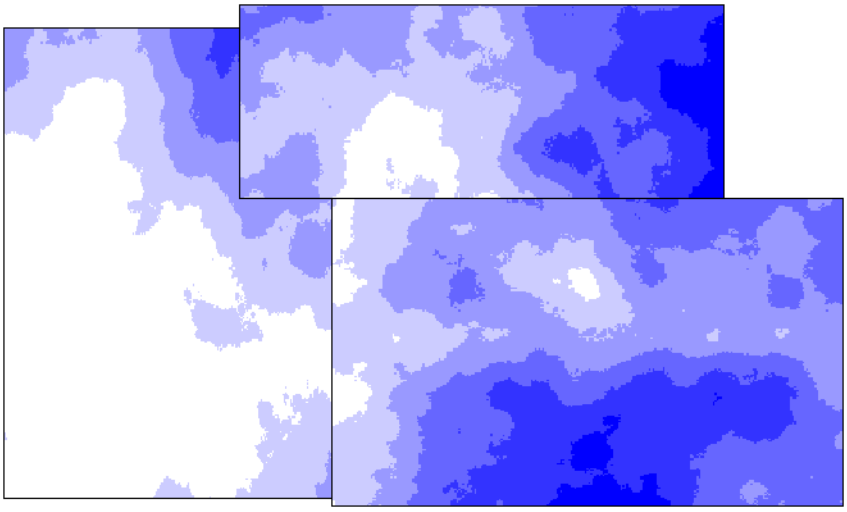
- Clouds, Smoke, and Vapor Trails
- Fire, Flares, and Explosions

Billboarding

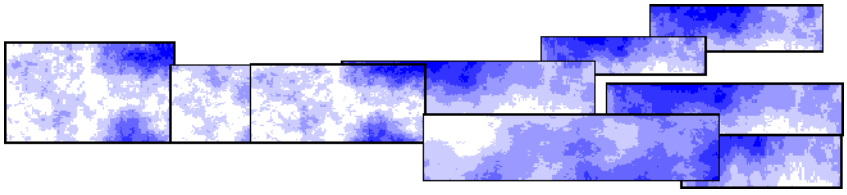
When creating 3-D scenes, an application can sometimes gain performance advantages by rendering 2-D objects in a way that makes them appear to be 3-D objects. This is the basic idea behind the technique of billboarding.
A billboard in the normal sense of the word is a sign along a roadway. Microsoft® Direct3D® applications can create and render this type of billboard by defining a rectangular solid and applying a texture to it. Billboarding in the more specialized sense of 3-D graphics is an extension of this. The goal is to make 2-D objects appear to be 3-D. The technique is to apply a texture that contains the object's image to a rectangular primitive. The primitive is rotated so that it always faces the user. It doesn't matter if the object's image is not rectangular. Portions of the billboard can be made transparent, so the parts of the billboard image that you don't want seen are not visible.
Many games employ billboarding for animated sprites. For instance, when the user is moving through a 3-D maze, he or she may see weapons or rewards that can be picked up. These are typically 2-D images textured on a rectangular primitive. Billboarding is often used in games to render images of trees, bushes, and clouds. When an image is applied to a billboard, the rectangular primitive must first be rotated so that the resulting image faces the user. Your application must then translate it into position. The application can then apply a texture to the primitive.
Billboarding works best for symmetrical objects, especially objects that are symmetrical along the vertical axis. It also requires that the altitude of the viewpoint doesn't increase too much. If the user is allowed to view the billboarded from above, it becomes readily apparent that the object is 2-D rather than 3-D.

Clouds, Smoke, and Vapor Trails

Clouds, smoke, and vapor trails can all be created by an extension of the billboarding technique. See Billboarding. By rotating the billboard on two axes instead of one, your application can enable the user to look at a billboard from any angle. Typically, an application rotates the billboard on the horizontal and vertical axes.
To make a simple cloud, your application can rotate a rectangular primitive on one or two axes so that the primitive faces the user. A cloud-like texture can then be applied to the primitive with transparency. For details on applying transparent textures to primitives, see Texture Blending. You can animate the cloud by applying a series of textures over time.
An application can create more complex clouds by forming them from a group of primitives. Each part of the cloud is a rectangular primitive. The primitives can be moved independently over time to give the appearance of a dynamic mist. This concept is illustrated in the following figure.



The appearance of smoke is displayed in a manner similar to clouds. It typically requires multiple billboards, like complex clouds. Smoke usually billows and rises over time, so the billboards that make up the smoke plume need to move accordingly. You may need to add more billboards as the plume rises and disperses.
A vapor trail is a smoke plume that doesn't rise. However, like a smoke plume, it disperses over time. The following figure illustrates the technique of using billboards to simulate a vapor trail.



Fire, Flares, and Explosions

You can use Microsoft® Direct3D® to simulate natural phenomena involving energy releases. For instance, an application can generate the appearance of fire by applying flame-like textures to a set of billboards. This is especially effective if the application uses a sequence of fire textures to animate the flames on each billboard in the fire. Varying the speed of the animation playback from billboard to billboard increases the appearance of real flames. The semblance of intermingled 3-D flames can be achieved by layering the billboards and the textures on the billboards.
You can simulate flares and flashes by applying successively brighter light maps to all primitives in a scene. Although this is a computationally high-overhead technique, it allows your application to simulate a localized flare or flash. That is, the portion of the scene where the flare or flash originates can brighten first.
Another technique is to position a billboard in front of the scene so that the entire render target area is covered. The application applies successively whiter textures to

the billboard and decreases the transparency over time. The entire scene fades to white as time passes. This is a low overhead method of creating a flare. However, using this technique, it can be difficult to generate the appearance of a bright flash from a single point light source.
Explosions can be displayed in a 3-D scene procedure similar to those used for fire, flashes, and flares. For instance, your application might use a billboard to display a shock wave and rising plume of smoke when the explosion occurs. At the same time, your application can use a set of billboards to simulate flames. In addition, it can position a single billboard in front of the scene to add a flare of light to the entire scene.
Energy beams can be simulated using billboards. Your application can also display them using primitives that are defined as line lists or line strips. For details, see Line Lists and Line Strips.
Your application can create force fields using billboards or primitives defined as triangle lists. To create a force field from triangle lists, define a set of disjoint triangles in a triangle list equally spaced over the region covered by the force field. The gaps between the triangles allow the user to see the scene behind the triangles, as you might expect when looking at a force field. Apply a texture to the triangle list that gives the triangles the appearance of glowing with energy. For further information, see Triangle Lists.

Antialiasing

Antialiasing is a technique you can use to reduce the appearance of stair-step pixels when drawing any line that is not exactly horizontal or vertical. In three-dimensional scenes, this artifact is most noticeable on the boundaries between polygons of different colors. Antialiasing effectively blends the pixels at these boundaries to produce a more natural look to the scene.
Microsoft® Direct3D® supports two antialiasing techniques: edge antialiasing and full-surface antialiasing. Which technique is best for your application depends on your requirements for performance and visual fidelity.

- Edge Antialiasing
- Full-Scene Antialiasing

The section below shows how to use special effects to simulate or use antialiasing.

- Motion Blur

Edge Antialiasing

In edge antialiasing, you render a scene, then re-render the convex silhouettes of the objects to antialias with lines. The system redraws those edges, blurring them to reduce artifacts.

[C++]
First, find out if the Microsoft® Direct3D® device supports antialiasing by calling the **IDirect3DDevice8::GetDeviceCaps** method. If a device supports edge antialiasing, **GetDeviceCaps** sets the D3DPRASTERCAPS_ANTIALIASEDGES capability flag in the **D3DCAPS8** structure to TRUE.

If the device supports antialiasing, set the D3DRS_EDGEANTIALIAS render state to TRUE, as shown in the code example below.

```
d3dDevice->SetRenderState( D3DRS_EDGEANTIALIAS, TRUE );
```

Now, redraw only the edges in the scene, using **IDirect3DDevice8::DrawPrimitive** and either the D3DPT_LINESTRIP or D3DPT_LINELIST primitive type. The behavior of edge antialiasing is undefined for primitives other than lines, so make sure to disable the feature by setting D3DRS_EDGEANTIALIAS to FALSE when antialiasing is complete.

[Visual Basic]
First, find out if the Microsoft® Direct3D® device supports antialiasing by calling the **Direct3D8.GetDeviceCaps** method. If a device supports edge antialiasing, **GetDeviceCaps** sets the D3DPRASTERCAPS_ANTIALIASEDGES flag the **D3DCAPS8** type under RasterCaps.
If the device supports antialiasing, set the D3DRS_EDGEANTIALIAS render state to TRUE, as shown in the code example below.

```
m_D3DDevice.SetRenderState D3DRS_EDGEANTIALIAS, TRUE
```

Now, redraw only the edges in the scene, using **Direct3DDevice8.DrawPrimitive** and either the D3DPT_LINESTRIP or D3DPT_LINELIST primitive type. The behavior of edge antialiasing is undefined for primitives other than lines, so make sure to disable the feature by setting D3DRENDERSTATE_EDGEANTIALIAS to FALSE when antialiasing is complete.

Redrawing every edge in your scene can work without introducing major artifacts, but it can be computationally expensive. In addition, it can be difficult to determine which edges should be antialiased. The most important edges to redraw are those between areas of very different colors, for example, silhouette edges, or boundaries between very different materials. Antialiasing the edge between two polygons of roughly the same color has no effect, yet is still computationally expensive. For these reasons, if current hardware supports full-scene antialiasing, it is often preferred. For more information, see Full-scene Antialiasing.

Full-Scene Antialiasing

Full-scene antialiasing refers to blurring the edges of each polygon in the scene as it is rasterized in a single pass—no second pass is required. Full-scene antialiasing, when supported, only affects triangles and groups of triangles; lines cannot be antialiased by using Microsoft® Direct3D® services. Full-scene antialiasing is done in Direct3D by using multisampling on each pixel. When multisampling is enabled all subsamples of a pixel are updated in one pass, but when used for other effects that involve multiple rendering passes, the application can specify that only some subsamples are to be affected by a given rendering pass. This latter approach enables simulation of motion blur, depth of field focus effects, reflection blur, and so on.

In both cases, the various samples recorded for each pixel are blended together and output to the screen. This enables the improved image quality of antialiasing or other effects.

[C++]

Before creating a device with the **IDirect3D8::CreateDevice** method, you need to determine if full-scene antialiasing is supported. Do this by calling the **IDirect3D8::CheckDeviceMultiSampleType** method as shown in the code example below.

```
/*
 * The code below assumes that pD3D is a valid pointer
 * to a IDirect3D8 interface.
 */

if( SUCCEEDED(pD3D->CheckDeviceMultiSampleType( D3DADAPTER_DEFAULT,
                                    D3DDEVTYPE_HAL , D3DFMT_R8G8B8,
                                    FALSE, D3DMULTISAMPLE_2_SAMPLES ) ) )
  // Full-scene antialiasing is supported. Enable it here.
```

The first parameter that **CheckDeviceMultiSampleType** accepts is an ordinal number that denotes the display adapter to query. This sample uses D3DADAPTER_DEFAULT to specify the primary display adapter. The second parameter is a value from the **D3DDEVTYPE** enumerated type, specifying the device type. The third parameter specifies the format of the surface. The fourth parameter tells Direct3D whether to inquire about full-windowed multisampling (TRUE) or full-scene antialiasing (FALSE). This sample uses FALSE to tell Direct3D that it is inquiring about full-scene antialiasing. The last parameter specifies the multisampling technique that you want to test. Use a value from the **D3DMULTISAMPLE_TYPE** enumerated type. This sample tests to see if two levels of multisampling are supported.

If the device supports the level of multisampling that you want to use, the next step is to set the presentation parameters by filling in the appropriate members of the **D3DPRESENT_PARAMETERS** structure to create a multisample rendering surface. After that, you can create the device. The sample code below shows how to set up a device with a multisampling render surface.

```
/*
 * The example below assumes that pD3D is a valid pointer
 * to a IDirect3D8 interface, d3dDevice is a pointer to a
 * IDirect3DDevice8 interface, and hWnd is a valid handle
 * to a window.
 */

D3DPRESENT_PARAMETER d3dPP
ZeroMemory( &d3dPP, sizeof( d3dPP ) );
d3dPP.Windowed        = FALSE
d3dPP.SwapEffect      = D3DSWAPEFFECT_DISCARD;
d3dPP.MultiSampleType = D3DMULTISAMPLE_2_SAMPLES;
```

```
pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
            D3DCREATE_SOFTWARE_VERTEXPROCESSING,
            &d3dpp, &d3dDevice)
```

Note

To use multisampling, the *SwapEffect* member of D3DPRESENT_PARAMETER must be set to D3DSWAPEFFECT_DISCARD.

The last step is to enable multisampling antialiasing by calling the **IDirect3DDevice8::SetRenderState** method and setting the D3DRS_MULTISAMPLEANTIALIAS to TRUE. After setting this value to TRUE, any rendering that you do will have multisampling applied to it. You might want to enable and disable multisampling, depending on what you are rendering.

[Visual Basic]

Before creating a device with the **Direct3D8.CreateDevice** method, you must determine if full-scene antialiasing is supported. Do this by calling the **Direct3D8.CheckDeviceMultiSampleType** method as shown in the code example below.

```
'
' The code below assumes that D3D is a valid pointer
' to a Direct3D8 object.
'

D3D->CheckDeviceMultiSampleType( D3DADAPTER_DEFAULT,
                    D3DDEVTYPE_HAL , D3DFMT_R8G8B8,
                    FALSE, D3DMULTISAMPLE_2_SAMPLES ) ) )
```

The first parameter that **CheckDeviceMultiSampleType** accepts is an ordinal number that denotes the display adapter to query. This sample uses D3DADAPTER_DEFAULT to specify the primary display adapter. The second parameter is a value from the **CONST_D3DDEVTYPE** enumerated type, specifying the device type. The third parameter specifies the format of the surface. The fourth parameter tells Direct3D whether to inquire about full-windowed multisampling (True) or full-scene antialiasing (False). This sample uses False to tell Direct3D that it is inquiring about full-scene antialiasing. The last parameter specifies the multisampling technique to test. Use a value from the **CONST_D3DMULTISAMPLE_TYPE** enumerated type. This sample tests to see if two levels of multisampling are supported.

If the device supports the level of multisampling that you want to use, the next step is to set the presentation parameters by filling in the appropriate members of the **D3DPRESENT_PARAMETERS** structure to create a multisample rendering surface. After that, you can create the device. The sample code below shows how to set up a device with a multisampling render surface.

```
'
' The example below assumes that pD3D is a valid Direct3D8
```

```
' object and D3DDevice is a valid Direct3DDevice8 object.
'

Dim d3dpp As D3DPRESENT_PARAMETER
d3dpp.Windowed = FALSE
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD
d3dpp.MultiSampleType = D3DMULTISAMPLE_2_SAMPLES
Set D3DDevice = pD3D->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
hWnd,
                    D3DCREATE_SOFTWARE_VERTEXPROCESSING, d3dpp )
```

Note

To use multisampling, the *SwapEffect* member of D3DPRESENT_PARAMETER must be set to D3DSWAPEFFECT_DISCARD.

The last step is to enable multisampling antialiasing by calling the **Direct3DDevice8.SetRenderState** method and setting the D3DRS_MULTISAMPLEANTIALIAS to True. After setting this value to True, any rendering that you do has multisampling applied to it. You might want to enable and disable multisampling depending on what you are rendering to the scene.

Motion Blur

You can enhance the perceived speed of an object in a 3-D scene by blurring the object and leaving a blurred trail of object images behind the object. Microsoft® Direct3D® applications accomplish this by rendering the object multiple times per frame.

[C++]
Recall that Direct3D applications typically render scenes into an off-screen buffer. The contents of the buffer are displayed on the screen when the application calls the **IDirect3DDevice8::Present** method. Your Direct3D application can render the object multiple times into a scene before it displays the frame on the screen.

[Visual Basic]
Recall that Direct3D applications typically render scenes into an off-screen buffer. The contents of the buffer are displayed on the screen when the application calls the **Direct3DDevice8.Present** method. Your Direct3D application can render the object multiple times into a scene before it displays the frame on the screen.

Programmatically, your application makes multiple calls to a **DrawPrimitive** method, repeatedly passing the same 3-D object. Before each call, the position of the object is updated slightly, producing a series of blurred object images on the target rendering surface. If the object has one or more textures, your application can enhance the motion blur effect by rendering the first image of the object with all its textures nearly

transparent. Each time the object renders, the transparency of the object's texture decreases. When your application renders the object in its final position, it should render the object's textures without transparency. The exception is if you're adding motion blur to another effect that requires texture transparency. In any case, the initial image of the object in the frame should be the most transparent. The final image should be the least transparent.
After your application renders the series of object images onto the target rendering surface and renders the rest of the scene, it should call the **Present** method to display the frame on the screen.
If your application is simulating the effect of the user moving through a scene at high speed, it can add motion blur to the entire scene. In this case, your application renders the entire scene multiple times per frame. Each time the scene renders, your application must move the viewpoint slightly. If the scene is highly complex, the user may see a visible performance degradation as acceleration is increased because of the increasing number of scene renderings per frame.

Bump Mapping

This section provides information about performing bump mapping with Microsoft® Direct3D®. The following topics introduce bump mapping, identify and define its key concepts, and provide details about how you can use bump-mapping in your applications.

- What Is Bump Mapping?
- Bump Map Pixel Formats
- Bump Mapping Formulas
- Using Bump Mapping

What Is Bump Mapping?

[C++]
Bump mapping is a special form of specular or diffuse environment mapping that simulates the reflections of finely tessellated objects without requiring extremely high polygon counts. The following image, based on the BumpEarth Sample, demonstrates bump mapping effects.

[Visual Basic]
Bump mapping is a special form of specular or diffuse environment mapping that simulates the reflections of finely tessellated objects without requiring extremely high polygon counts. The following image, based on the C++ BumpEarth Sample, demonstrates bump mapping effects.

The globe on the left is a sphere textured with an image of the Earth's surface, with a specular environment map applied. The globe on the right is exactly the same, but also has a bump map applied. The polygon count for the second globe is unchanged from that of the first globe.
Bump mapping in Microsoft® Direct3D® can be accurately described as per-pixel texture coordinate perturbation of specular or diffuse environment maps, because you provide information about the contour of the bump map in terms of delta-values, which the system applies to the *u* and *v* texture coordinates of an environment map in the next texture stage. The delta values are encoded in the pixel format of the bump map surface. For more information, see Bump Map Pixel Formats.
The BumpEarth sample uses a height-map to store contour data. When the sample starts, it processes the pixels in the height map to compute the appropriate *u* and *v* delta values, based on the relative height of each pixel to the four neighboring pixels.

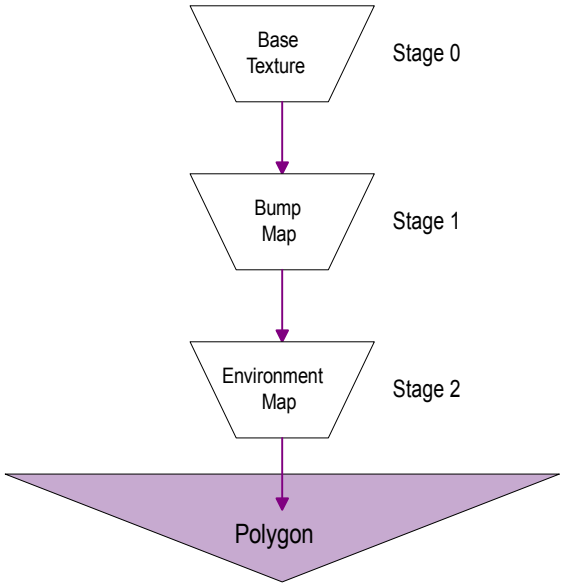
Note

[C++]
Direct3D does not natively support height-maps; they are merely a convenient format in which to store and visualize contour data. The preceding image is based on the height map used by the BumpEarth sample to produce the bump map it supplies to Direct3D. and is included here for descriptive purposes. Your application can store contour information in any format, or even generate a procedural bump map, as the BumpWaves Sample does.

[Visual Basic]
Direct3D does not natively support height-maps; they are merely a convenient format in which to store and visualize contour data. The preceding image is based on the height map used by the BumpEarth sample to produce the bump map it supplies to Direct3D. and is included here for descriptive purposes. Your application can store contour information in any format, or even generate a procedural bump map, as the C++ BumpWaves Sample does.

This is the height map image that the BumpEarth sample uses to compute the delta values encoded in the pixel format of the bump map texture.

Bump mapping relies completely on multiple texture blending services, and requires that the device support at least two blending stages—one for the bump map, and another for an environment map. A minimum of three texture blending stages are required to apply a base texture map, the most common case. The following figure shows the relationships between the base texture, the bump map, and the environment map in the texture blending cascade.



You must prepare the texture stages appropriately to enable bump mapping. This task is the topic of Configuring Bump Mapping Parameters.

Bump Map Pixel Formats

A bump map is a IDirect3DTexture8 object that uses a specialized pixel format. Rather than storing red, green, and blue color components, each pixel in a bump map stores the delta values for *u* and *v* (D_u and D_v) and sometimes a luminance component, *L*. These values are applied by the system as described in the Bump Mapping Formulas topic.

[C++]
You can specify a bump map pixel format by setting the **D3DFORMAT** enumerated type to one of the following bump map pixel formats.

Format	Description
D3DFMT_V8U8	16-bit bump-map format.
D3DFMT_L6V5U5	16-bit bump-map format with luminance.
D3DFMT_X8L8V8U8	32-bit bump-map format with luminance where 8 bits are reserved for each element.
D3DFMT_Q8W8V8U8	32-bit bump-map format.
D3DFMT_V16U16	32-bit bump-map format.
D3DFMT_W11V11U10	32-bit bump-map format.

The D_u and D_v components of a pixel are signed values that range from –1.0 to +1.0. The luminance component, when used, is an unsigned integer value that ranges from 0 to 255.

[Visual Basic]
You can specify a bump map pixel format by setting the **CONST_D3DFORMAT** enumerated type to one of the following bump map pixel formats.

Format	Description
D3DFMT_V8U8	16-bit bump-map format.
D3DFMT_L6V5U5	16-bit bump-map format with luminance.
D3DFMT_X8L8V8U8	32-bit bump-map format with luminance where 8 bits are reserved for each element.
D3DFMT_Q8W8V8U8	32-bit bump-map format.
D3DFMT_V16U16	32-bit bump-map format.
D3DFMT_W11V11U10	32-bit bump-map format.

The D_u and D_v components of a pixel are signed values that range from –1.0 to +1.0. The luminance component, when used, is an unsigned integer value that ranges from 0 to 255.

Note
Before picking a bump map pixel format, you should check if the particular format is supported. For more information, see Detecting Support for Bump Mapping.

Bump Mapping Formulas

[C++]
Microsoft® Direct3D® applies the following formulas to the D_u and D_v components in each bump map pixel.

$$D_u' = D_u M_{0,0} + D_v M_{1,0}$$

$$D_v' = D_u M_{0,1} + D_v M_{1,1}$$

In these formulas, the D_u and D_v variables are taken directly from a bump map pixel and transformed by a 2×2 matrix to produce the output delta values D_u' and D_v'. The system uses the output values to modify the texture coordinates that address the environment map in the next texture stage. The coefficients of the transformation matrix are set though the D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT10, D3DTSS_BUMPENVMAT01, and D3DTSS_BUMPENVMAT11 texture stage states.
In addition to the *u* and *v* delta values, the system can compute a luminance value that it uses to modulate the color of the environment map in the next blending stage.

$$L' = LS + O$$

In this formula, L' is the output luminance being computed. The L variable is the luminance value taken from a bump map pixel, which is multiplied by the scaling factor, S, and offset by the value in the variable O. The D3DTSS_BUMPENVLSCALE and D3DTSS_BUMPENVLOFFSET texture stage states control the values for the S and O variables in the formula. This formula is only applied when the texture blending operation for the stage that contains the bump map is set to D3DTOP_BUMPENVMAPLUMINANCE. When using D3DTOP_BUMPENVMAP, the system uses a value of 1.0 for L'.
After computing the output delta values D_u' and D_v', the system adds them to the texture coordinates in the next texture stage, and modulates the chosen color by the luminance to produce the color applied to the polygon.

[Visual Basic]
Microsoft® Direct3D® applies the following formulas to the D_u and D_v components in each bump map pixel.

$$D_u' = D_u M_{0,0} + D_v M_{1,0}$$

$$D_v' = D_u M_{0,1} + D_v M_{1,1}$$

In these formulas, the D_u and D_v variables are taken directly from a bump map pixel and transformed by a 2×2 matrix to produce the output delta values D_u' and D_v'. The system uses the output values to modify the texture coordinates that address the environment map in the next texture stage. The coefficients of the transformation matrix are set though the D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT10, D3DTSS_BUMPENVMAT01, and D3DTSS_BUMPENVMAT11 texture stage states.
In addition to the u and v delta values, the system can compute a luminance value that it uses to modulate the color of the environment map in the next blending stage.

$$L' = LS + O$$

In this formula, L' is the output luminance being computed; L' is clamped to the range of 0.0 to 1.0, inclusive. The L variable is the luminance value taken from a bump map pixel, which is multiplied by the scaling factor, S, and offset by the value in the variable O. The D3DTSS_BUMPENVLSCALE and D3DTSS_BUMPENVLOFFSET texture stage states control the values for the S and O variables in the formula. This formula is only applied when the texture blending operation for the stage that contains the bump map is set to D3DTOP_BUMPENVMAPLUMINANCE. When using D3DTOP_BUMPENVMAP, the system uses a value of 1.0 for L'.

After computing the output delta values D_u' and D_v', the system adds them to the texture coordinates in the next texture stage, and modulates the chosen color by the luminance to produce the color applied to the polygon.

Using Bump Mapping

The following topics provide details about the most common tasks an application must perform when using Microsoft® Direct3D® bump mapping.

- Detecting Support for Bump Mapping
- Creating a Bump Map Texture
- Configuring Bump Mapping Parameters

Detecting Support for Bump Mapping

[C++]
A Microsoft® Direct3D® device can perform bump mapping if it supports either the D3DTOP_BUMPENVMAP or D3DTOP_BUMPENVMAPLUMINANCE texture blending operation. Additionally, applications should check the device capabilities to make sure the device supports an appropriate number of blending stages, usually at least three, and exposes at least one bump-mapping pixel format.
The following code example checks device capabilities to detect support for bump mapping in the current device, using the given criteria.

```
BOOL SupportsBumpMapping()
{
    D3DCAPS8 d3dCaps;

    d3dDevice->GetDeviceCaps( &d3dCaps );

    // Does this device support the two bump mapping blend operations?
    if ( 0 == d3dCaps.TextureOpCaps & ( D3DTEXOPCAPS_BUMPENVMAP |
                                D3DTEXOPCAPS_BUMPENVMAPLUMINANCE ))
        return FALSE;

    // Does this device support up to three blending stages?
    if( d3dCaps.MaxTextureBlendStages < 3 )
        return FALSE;
}
```

[Visual Basic]
A Microsoft® Direct3D® device can perform bump mapping if it supports either the D3DTOP_BUMPENVMAP or D3DTOP_BUMPENVMAPLUMINANCE texture blending operation. Additionally, applications should check the device capabilities to

make sure the device supports an appropriate number of blending stages, usually at least three, and exposes at least one bump mapping pixel format.
The following code checks device capabilities to detect support for bump mapping in the current device, using the given criteria.

```
Function SupportsBumpMapping As Bool
{
    Dim d3dCaps As D3DCAPS8

    m_D3DDevice.GetDeviceCaps d3dCaps

    ' Does this device support the two bump mapping blend operations?
    If ( d3dCaps.TextureOpsCaps And _
        ( D3DTEXOPCAPS_BUMPENVMAP Or D3DTEXOPCAPS_BUMPENVMAPLUMINANCE
) = 0) Then
        SupportBumpMapping = FALSE
    End If

    // Does this device support up to three blending stages?
    If d3dCaps.MaxTextureBlendStages < 3 Then
       SupportBumpMapping = FALSE
    End If

    SupportBumpMapping = TRUE
}
```

Creating a Bump Map Texture

You create a bump map texture like any other texture. Once your application verifies support for bump mapping and retrieves a valid pixel format, as discussed in Detecting Support For Bump Mapping, you can create a bump map texture surface. For information on creating textures, see Obtaining a Texture Surface Object.
After the surface is created, you can load each pixel with the appropriate delta values, and luminance, if the surface format includes luminance. The values for each pixel component are described in Bump Map Pixel Formats.

Configuring Bump Mapping Parameters

When your application has created a bump map and set the contents of each pixel, you can prepare for rendering by configuring bump mapping parameters. Bump mapping parameters include setting the required textures and their blending operations, as well as the transformation and luminance controls for the bump map itself.

[C++]

To configure bump mapping parameters in C++

1. Set the base texture map if used, bump map, and environment map textures into texture blending stages.

2. Set the color and alpha blending operations for each texture stage.
3. Set the bump map transformation matrix.
4. Set the luminance scale and offset values as needed.

The following code example sets three textures—the base texture map, the bump map, and a specular environment map—to the appropriate texture blending stages.

```
// Set the three textures.
d3dDevice->SetTexture( 0, d3dBaseTexture );
d3dDevice->SetTexture( 1, d3dBumpMap );
d3dDevice->SetTexture( 2, d3dEnvMap );
```

After setting the textures to their blending stages, the following code example prepares the blending operations and arguments for each stage.

```
// Set the color operations and arguments to prepare for
// bump mapping.

// Stage 0: the base texture.
d3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
d3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
d3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1 );
d3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );

// Stage 1: the bump map - Use luminance for this example.
d3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
d3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,
D3DTOP_BUMPENVMAPLUMINANCE);
d3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );

// Stage 2: a specular environment map.
d3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX, 0 );
d3dDevice->SetTextureStageState( 2, D3DTSS_COLOROP, D3DTOP_ADD );
d3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG1, D3DTA_TEXTURE );
d3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG2, D3DTA_CURRENT );
```

Once the blending operations and arguments are set, the following code example sets the 2×2 bump mapping matrix to the identity matrix by setting the D3DTSS_BUMPENVMAT00 and D3DTSS_BUMPENVMAT11 texture stage states to 1.0. Setting the matrix to the identity causes the system to use the delta-values in the bump map unmodified, but this is not a requirement.

```
// Set the bump mapping matrix.
//
// NOTE
//    These calls rely on the following inline shortcut function:
```

```
//      inline DWORD F2DW( FLOAT f ) { return *((DWORD*)&f); }
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT00, F2DW(1.0f) );
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT01, F2DW(0.0f) );
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT10, F2DW(0.0f) );
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT11, F2DW(1.0f) );
```

If you set the bump mapping operation to include luminance (D3DTOP_BUMPENVMAPLUMINANCE), you must set the luminance controls. The luminance controls configure how the system computes luminance before modulating the color from the texture in the next stage. For details, see Bump Mapping Formulas.

```
// Set luminance controls. This is only needed when using
// a bump map that contains luminance, and when the
// D3DTOP_BUMPENVMAPLUMINANCE texture blending operation is
// being used.
//
// NOTE
//    These calls rely on the following inline shortcut function:
//      inline DWORD F2DW( FLOAT f ) { return *((DWORD*)&f); }
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLSCALE, F2DW(0.5f) );
d3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLOFFSET, F2DW(0.0f) );
```

After your application configures bump mapping parameters, it can render as normal, and the rendered polygons receive bump mapping effects.

[Visual Basic]

Ù To configure bump mapping parameters in Visual Basic

1. Set the base texture map if used, bump map, and environment map textures into texture blending stages.
2. Set the color and alpha blending operations for each texture stage.
3. Set the bump map transformation matrix.
4. Set the luminance scale and offset values, as needed.

The following code example sets three textures—the base texture map, the bump map, and a specular environment map—to the appropriate texture blending stages.

```
' Set the three textures.
d3dDevice.SetTexture(0, d3dBaseTexture)
d3dDevice.SetTexture(1, d3dBumpMap)
d3dDevice.SetTexture(2, d3dEnvMap)
```

After setting the textures to their blending stages, the following code example prepares the blending operations and arguments for each stage.

```
' Create the bump map texture.
' Set the color operations and arguments to prepare for bump mapping.
```

```
' Stage 0: the base texture.
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE)
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE)
Call d3dDevice.SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1)
Call d3dDevice.SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE)
Call d3dDevice.SetTextureStageState(0, D3DTSS_TEXCOORDINDEX, 1)

' Stage 1: the bump map - Use luminance for this example.
Call d3dDevice.SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 1)
Call d3dDevice.SetTextureStageState(1, D3DTSS_COLOROP,
D3DTOP_BUMPENVMAPLUMINANCE)
Call d3dDevice.SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDevice.SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT)

' Stage 2: a specular environment map.
Call d3dDevice.SetTextureStageState(2, D3DTSS_TEXCOORDINDEX, 0)
Call d3dDevice.SetTextureStageState(2, D3DTSS_COLOROP, D3DTOP_ADD)
Call d3dDevice.SetTextureStageState(2, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDevice.SetTextureStageState(2, D3DTSS_COLORARG2, D3DTA_CURRENT)
```

Once the blending operations and arguments are set, the following code example sets the 2×2 bump mapping matrix to the identity matrix, by setting the D3DTSS_BUMPENVMAT00 and D3DTSS_BUMPENVMAT11 texture stage states to 1.0. Setting the matrix to the identity causes the system to use the delta-values in the bump map unmodified, but this is not a requirement.

```
' Set the bump mapping matrix.
Call d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVMAT00, 1#)
Call d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVMAT01, 0#)
Call d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVMAT10, 0#)
Call d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVMAT11, 1#)
```

If you set the bump mapping operation to include luminance (D3DTOP_BUMPENVMAPLUMINANCE), you must set the luminance controls. The luminance controls configure how the system computes luminance before modulating the color from the texture in the next stage. For details, see Bump Mapping Formulas.

```
' Set luminance controls. This is only needed when using a bump map
' that contains luminance, and when the D3DTOP_BUMPENVMAPLUMINANCE
' texture blending operation is being used.
Call d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVLSCALE, 0.5)
Call d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVLOFFSET, 0#)
```

After your application configures bump mapping parameters, it can render as normal, and the rendered polygons receive bump mapping effects.

Note

The preceding example shows parameters set for specular environment mapping. When performing diffuse light mapping, applications set the texture blending operation for the last stage to D3DTOP_MODULATE. For more information, see Diffuse Light Maps.

Environment Mapping

This section provides information about performing two common types of environment mapping with Microsoft® Direct3D®. There are many types of environment mapping in use throughout the graphics industry, but the following topics target the two most common forms: cubic environment mapping and spherical environment mapping.

- What Is Environment Mapping?
- Cubic Environment Mapping
- Spherical Environment Mapping

What Is Environment Mapping?

Environment mapping is a technique that simulates highly reflective surfaces without using ray-tracing. In practice, environment mapping applies a special texture map that contains an image of the scene surrounding an object to the object itself. The result approximates the appearance of a reflective surface, close enough to fool the eye, without incurring any of the expensive computations involved in ray-tracing.

[C++]
The following image is taken from the SphereMap Sample, which, as its name implies, uses a type of environment mapping called spherical environment mapping. For details, see Spherical Environment Mapping.

[Visual Basic]
The following image is taken from the C++ SphereMap Sample, which, as its name implies, uses a type of environment mapping called spherical environment mapping. For details, see Spherical Environment Mapping.

The teapot in this image appears to reflect its surroundings; this is actually a texture being applied to the object. Because environment mapping uses a texture, combined with specially computed texture coordinates, it can be performed in real-time.

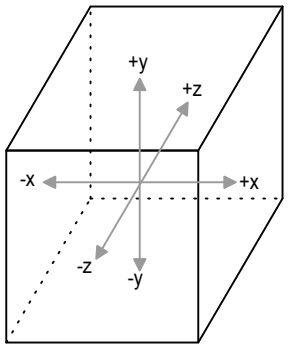
Cubic Environment Mapping

Cubic environment mapping textures an object with an image of the space or lighting effects surrounding the object, represented in the form of six internal faces of a cube. Information in this section is divided into the following topics.

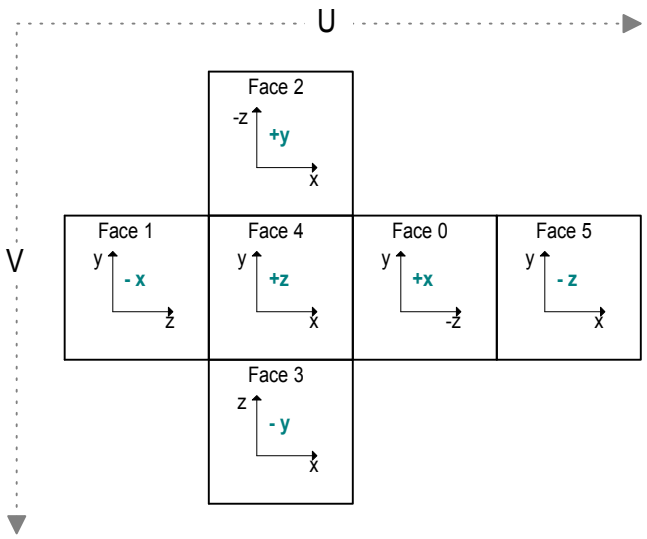
- What Are Cubic Environment Maps?
- Creating Cubic Environment Map Surfaces
- Accessing Cubic Environment Map Faces
- Mipmapped Cubic Environment Maps
- Rendering to Cubic Environment Maps
- Texture Coordinates for Cubic Environment Maps

What Are Cubic Environment Maps?

Cubic environment maps—sometimes casually referred to as cube maps—are textures that contain image data representing the scene surrounding an object, as if the object were in the center of a cube. Each face of the cubic environment map covers a 90-degree field of view in the horizontal and vertical, and there are six faces per cube map. The orientation of the faces is given in the following illustration.



Each face of the cube is oriented perpendicular to the x/y, y/z, or x/z plane, in world space. The following figure shows how each plane corresponds to a face.



Cubic-environment maps are implemented as a series of texture objects. Applications can use static images for cubic-environment mapping, or they can render into the faces of the cube map to perform dynamic environment mapping. This technique requires that the cube-map surfaces be valid render-target surfaces, created with the D3DUSAGE_RENDERTARGET flag set.
The faces of a cube map don't need to contain extremely detailed renderings of the surrounding scene. In most cases, environment maps are applied to curved surfaces. Given the amount of curvature used by most applications, the resulting reflective distortion makes extreme detail in the environment map wasteful in terms of memory and rendering overhead.

Creating Cubic-Environment Map Surfaces

[C++]
You create a cubic environment map texture by calling the **IDirect3DDevice8::CreateCubeTexture** method. Cubic-environment map textures must be square, with dimensions that are a power of two.
The following code example shows how your C++ application might create a simple cubic-environment map.

```
/*
 * For this example, m_d3dDevice is a valid
 * pointer to an IDirect3DDevice8 interface.
 */

LPDIRECT3DCUBETEXTURE8 m_pCubeMap;

m_d3dDevice->CreateCubeTexture( 256, 1, D3DUSAGE_RENDERTARGET,
D3DFMT_R8G8B8,
                    D3DPOOL_MANAGED, &m_pCubeMap );
```

[Visual Basic]
You create a cubic environment map texture by calling the **D3DX8.CreateCubeTexture** method. Cubic-environment map textures must be square, with dimensions that are a power of two.
The following code shows how your Microsoft® Visual Basic® application might create a simple cubic-environment map.

```
'
' For this example, d3dDevice is a valid Direct3DDevice8
' object and g_D3DX is a valid D3DX object.
'
Dim CubeMap As Direct3DCubeTexture8 CubeTexture

Set CubeMap = g_D3DX.CreateCubeTexture( d3dDevice, 256, 1,
D3DUSAGE_RENDERTARGET, _
                              D3DFMT_R8G8B8, D3DPOOL_MANAGED );
```

Accessing Cubic Environment Map Faces

[C++]
You can navigate between faces of a cubic environment map by using the **IDirect3DCubeTexture8::GetCubeMapSurface** method.
The following code example uses **GetCubeMapSurface** to retrieve the cube-map surface used for the positive-y face (face 2).

```
/*
 * For this example, m_pCubeMap is a valid
 * pointer to a IDirect3DCubeTexture8 interface.
 */
LPDIRECT3DSURFACE8 pFace2;

m_pCubeMap->GetCubeMapSurface( D3DCUBEMAP_FACE_POSITIVE_Y, 0, &pFace2);
```

The first parameter that **GetAttachedSurface** accepts is a **D3DCUBEMAP_FACES** enumerated value that describes the attached surface that the method should retrieve. The second parameter tells Microsoft® Direct3D® which level of a mipmapped cube texture to retrieve. The third parameter accepted is the address of the **IDirect3DSurface8** interface, representing the returned cube texture surface. Because this cube-map is not mipmapped, 0 is used here.

Note

After calling this method, the internal reference count on the **IDirect3DSurface8** interface is increased. When you are done using this surface, be sure to call the **IUnknown::Release** method on this **IDirect3DSurface8** interface or you will have a memory leak.

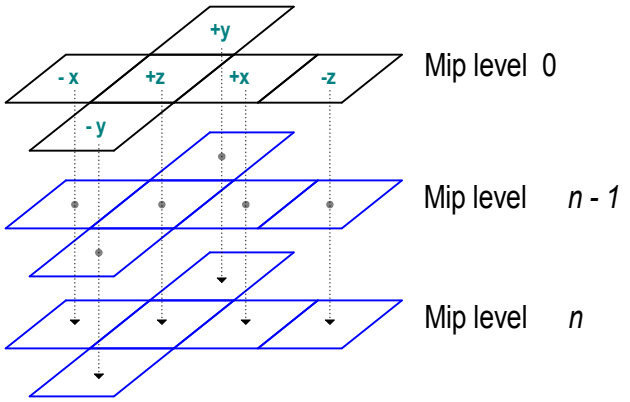
[Visual Basic]
You can navigate between faces of a cubic environment map by using the **Direct3DCubeTexture8.GetCubeMapSurface** method.
The following code example uses **GetCubeMapSurface** to retrieve the cube-map surface used for the positive-y face (face 2).

```
'
' For this example, CubeMap is a valid
' Direct3DCubeTexture8 object.
'
Dim Face2 As Direct3DSurface8

Set Face2 = CubeMap.GetCubeMapSurface( D3DCUBEMAP_FACE_POSITIVE_Y, 0 );
```

The first parameter that **GetAttachedSurface** accepts is a **CONST_D3DCUBEMAP_FACES** enumerated value that describes the attached surface that the method should retrieve. The second parameter tells Microsoft® Direct3D® which level of a mipmapped cube texture to retrieve. Because this cube-map is not mipmapped, 0 is used here.

Mipmapped Cubic Environment Maps

Cube maps can be mipmapped. If you include the DDSCAPS_MIPMAP capability, the system creates each face with attached mipmap surfaces. You can envision the topography of these surfaces as shown in the following graphic.



[C++]
Applications that create mipmapped cubic-environment maps can access each face by calling the **IDirect3DCubeTexture8::GetCubeMapSurface** method. Start by setting

the appropriate value from the D3DCUBEMAP_FACES enumerated type, as discussed in Accessing Cubic Environment Map Faces. Next, select the level to retrieve by setting the *Level* parameter to the mipmap level that you want. Remember that 0 corresponds with the top-level image.

[Visual Basic]
Applications that create mipmapped cubic-environment maps can access each face by calling the **Direct3DCubeTexture8.GetCubeMapSurface** method. Start by setting the appropriate value from the D3DCUBEMAP_FACES enumerated type, as discussed in Accessing Cubic Environment Map Faces. Next, select the level to retrieve by setting the *Level* parameter to the mipmap level that you want. Remember that 0 corresponds with the top-level image.

Rendering to Cubic Environment Maps

You can copy images to the individual faces of the cube map just like you would any other texture or surface object. The most important thing to do before rendering to a face is set the transformation matrices so that the camera is positioned properly and points in the proper direction for that face: forward (+z), backward (-z), left (-x), right (+x), up (+y), or down (-y).

[C++]
The following C++ code example prepares and sets a view matrix according to the face being rendered.

```
/*
 * For this example, pCubeMap is a valid pointer to a
 * IDirect3DCubeTexture8 interface and d3dDevice is a
 * valid pointer to a IDirect3DDevice8 interface.
 */
void RenderFaces()
{
    // Save transformation matrices of the device.
    D3DXMATRIX matProjSave, matViewSave;
    d3dDevice->GetTransform( D3DTS_VIEW,       &matViewSave );
    d3dDevice->GetTransform( D3DTS_PROJECTION, &matProjSave );

    // Store the current back buffer and z-buffer.
    LPDIRECT3DSURFACE8 pBackBuffer, pZBuffer;
    d3dDevice->GetRenderTarget( &pBackBuffer );
    d3dDevice->GetDepthStencilSurface( &pZBuffer );
```

Remember, each face of a cubic-environment map represents a 90-degree field of view. Unless your application requires a different field of view angle—for special effects, for example—take care to set the projection matrix accordingly.
This code example creates and sets a projection matrix for the most common case.

```
    // Use 90-degree field of view in the projection.
```

```
D3DMATRIX matProj;
D3DXMatrixPerspectiveFovLH( matProj, D3DX_PI/2, 1.0f, 0.5f, 1000.0f );
d3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );

// Loop through the six faces of the cube map.
for( DWORD i=0; i<6; i++ )
{
   // Standard view that will be overridden below.
   D3DVECTOR vEnvEyePt = D3DVECTOR( 0.0f, 0.0f, 0.0f );
   D3DVECTOR vLookatPt, vUpVec;

   switch( i )
   {
      case D3DCUBEMAP_FACE_POSITIVE_X:
         vLookatPt = D3DVECTOR( 1.0f, 0.0f, 0.0f );
         vUpVec    = D3DVECTOR( 0.0f, 1.0f, 0.0f );
         break;
      case D3DCUBEMAP_FACE_NEGATIVE_X:
         vLookatPt = D3DVECTOR(-1.0f, 0.0f, 0.0f );
         vUpVec    = D3DVECTOR( 0.0f, 1.0f, 0.0f );
         break;
      case D3DCUBEMAP_FACE_POSITIVE_Y:
         vLookatPt = D3DVECTOR( 0.0f, 1.0f, 0.0f );
         vUpVec    = D3DVECTOR( 0.0f, 0.0f,-1.0f );
         break;
      case D3DCUBEMAP_FACE_NEGATIVE_Y:
         vLookatPt = D3DVECTOR( 0.0f,-1.0f, 0.0f );
         vUpVec    = D3DVECTOR( 0.0f, 0.0f, 1.0f );
         break;
      case D3DCUBEMAP_FACE_POSITIVE_Z:
         vLookatPt = D3DVECTOR( 0.0f, 0.0f, 1.0f );
         vUpVec    = D3DVECTOR( 0.0f, 1.0f, 0.0f );
         break;
      case D3DCUBEMAP_FACE_NEGATIVE_Z:
         vLookatPt = D3DVECTOR( 0.0f, 0.0f,-1.0f );
         vUpVec    = D3DVECTOR( 0.0f, 1.0f, 0.0f );
         break;
   }

   D3DMATRIX matView;
   D3DXMatrixLookAtLH( matView, vEnvEyePt, vLookatPt, vUpVec );
   d3dDevice->SetTransform( D3DTS_VIEW, &matView );
```

Once the camera is in position and the projection matrix set, you can render the scene. Each object in the scene should be positioned as you would normally position them. The following code example, provided for completeness, outlines this task.

```
        //Get pointer to surface in order to render to it.
        LPDIRECT3DSURFACE8 pFace;
        pCubeMap->GetCubeMapSurface( (D3DCUBEMAP_FACES)i, 0, &pFace );
        d3dDevice->SetRenderTarget ( pFace , pZBuffer );
        pFace->Release();

        d3dDevice->BeginScene();
        // Render scene here.
        d3dDevice->EndScene();
    }

    // Change the render target back to the main back buffer.
    d3dDevice->SetRenderTarget( pBackBuffer, pZBuffer );
    pBackBuffer->Release();
    pZBuffer->Release();

    // Restore the original transformation matrices.
    d3dDevice->SetTransform( D3DTS_VIEW,       &matViewSave );
    d3dDevice->SetTransform( D3DTS_PROJECTION, &matProjSave );
}
```

Note the call to the **IDirect3DDevice8::SetRenderTarget** method. When rendering to the cube map faces, you must assign the face as the current render-target surface. Applications that use depth buffers can explicitly create a depth-buffer for the render-target, or reassign an existing depth-buffer to the render-target surface. The code sample above uses the latter approach.

[Visual Basic]

The following Microsoft® Visual Basic® code example prepares and sets a view matrix according to the face being rendered.

```
'
' For this example, CubeMap is a valid Direct3DCubeTexture8 object
' and m_D3DDevice is a valid Direct3DDevice8 object.
'
Sub RenderFaces

    ' Save transformation matrices of the device.
    Dim matProjSave As D3DMATRIX, _
        matViewSave As D3DMATRIX
    m_D3DDevice.GetTransform D3DTS_VIEW, matViewSave
    m_D3DDevice.GetTransform D3DTS_PROJECTION, matProjSave

    ' Store the current back buffer and z-buffer.
    Dim BackBuffer As Direct3DSurface8, _
        Zbuffer As Direct3DSurface8
```

```
Set BackBuffer = m_D3DDevice.GetRenderTarget()
Set Zbuffer = m_D3DDevice.GetDepthStencilSurface()
```

Remember, each face of a cubic-environment map represents a 90-degree field of view. Unless your application requires a different field of view angle—for special effects, for example—take care to set the projection matrix accordingly.
This code example creates and sets a projection matrix for the most common case.

```
' Use 90-degree field of view in the projection.
Dim matProj As D3DMATRIX
D3DXMatrixPerspectiveFovLH matProj, g_pi/4, 1, 1/2, 1000
m_D3DDevice.SetTransform D3DTS_PROJECTION, matProj

' Loop through the six faces of the cube map.
Dim i As Integer
For i = 0 To 6

   'Standard view that will be overridden below.
   Dim vEnvEyePt As D3DVECTOR, _
      vLookatPt As D3DVECTOR, _
      vUpVec As D3DVECTOR

   Select Case i

      case D3DCUBEMAP_FACE_POSITIVE_X:
         vLookatPt.x = 1#: vUpVec.y = 1#

      case D3DCUBEMAP_FACE_NEGATIVE_X:
         vLookatPt.x = -1#: vUpVec.y = 1#

      case D3DCUBEMAP_FACE_POSITIVE_Y:
         vLookatPt.y = 1#: vUpVec.z = -1#

      case D3DCUBEMAP_FACE_NEGATIVE_Y:
         vLookatPt.y = -1#: vUpVec.z = 1#

      case D3DCUBEMAP_FACE_POSITIVE_Z:
         vLookatPt.z = 1#: vUpVec.y = 1#

      case D3DCUBEMAP_FACE_NEGATIVE_Z:
         vLookatPt.z = -1#: vUpVec.y = 1#

   End Select

   Dim matView As D3DMATRIX
   D3DXMatrixLookAtLH matView, vEnvEyePt, vLookatPt, vUpVec
   m_D3DDevice.SetTransform D3DTS_VIEW, matView
```

Once the camera is in position and the projection matrix set, you can render the scene. Each object in the scene should be positioned as you would normally position them. The following code example, provided for completeness, outlines this task.

```
        ' Get cube surface in order to render to it.
        Dim Face As Direct3DSurface8
        Set Face = CubeMap.GetCubeMapSurface i, 0
        m_D3DDevice.SetRenderTarget Face, Zbuffer
        Set Face = Nothing

        m_D3DDevice.BeginScene
        ' Render scene here.
        m_D3DDevice.EndScene

    Next

    ' Change the render target back to the main back buffer.
    m_D3DDevice.SetRenderTarget BackBuffer, pZBuffer
    Set BackBuffer = Nothing
    Set Zbuffer = Nothing

    // Restore the original transformation matrices.
    m_D3DDevice.SetTransform D3DTS_VIEW, matViewSave
    m_D3DDevice.SetTransform D3DTS_PROJECTION, matProjSave

End Sub
```

Note the call to the **Direct3DDevice8.SetRenderTarget** method. When rendering to the cube map faces, you must assign the face as the current render-target surface. Applications that use depth buffers can explicitly create a depth-buffer for the render-target, or reassign an existing depth-buffer to the render-target surface. The code sample above uses the latter method.

Texture Coordinates for Cubic Environment Maps

[C++]

Texture coordinates that index a cubic-environment map aren't simple u, v style coordinates, as used when standard textures are applied. In fact, cubic environment maps don't use texture coordinates at all. In place of a set of texture coordinates, cubic environment maps require a 3-D vector. You must take care to specify a proper vertex format. In addition to telling the system how many sets of texture coordinates your application uses, you must provide information about how many elements are in each set. Microsoft® Direct3D® offers the **D3DFVF_TEXCOORDSIZEn** set of macros for this purpose. These macros accept a single parameter, identifying the index of the texture coordinate set for which the size is being described. In the case of a 3-D

vector, you include the bit pattern created by the D3DFVF_TEXCOORDSIZE3 macro. The following code example shows how this macro is used.

```
/*
 * Create a flexible vertex format descriptor for a vertex that
 * contains a position, normal, and one set of 3-D texture
 * coordinates.
 */
DWORD dwFVF = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1 |
D3DFVF_TEXCOORDSIZE3(0);
```

In some cases, such as diffuse light mapping, you use the camera-space vertex normal for the vector. In other cases, like specular environment mapping, you use a reflection vector. Because transformed vertex normals are widely understood, the information here concentrates on computing a reflection vector.

Computing a reflection vector on your own requires understanding of the position of each vertex, and of a vector from the viewpoint to that vertex. Direct3D can automatically compute the reflection vectors for your geometry. Using this feature saves memory because you don't need to include texture coordinates for the environment map. It also reduces bandwidth and, in the case of a TnLHAL Device, it can be significantly faster than the computations that your application can make on its own. To use this feature, in the texture stage that contains the cubic-environment map, set the D3DTSS_TEXCOORDINDEX texture stage state to a combination of the D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR flag and the index of a texture coordinate set. In some situations, like diffuse light mapping, you might use the D3DTSS_TCI_CAMERASPACENORMAL flag, which causes the system to use the transformed, camera-space, vertex normal as the addressing vector for the texture. The index is only used by the system to determine the wrapping mode for the texture. The following code example shows how this value is used.

```
/*
 * The m_d3dDevice variable is a valid pointer
 * to an IDirect3DDevice8 interface.
 *
 * Automatically generate texture coordinates for stage 2.
 * This assumes that stage 2 is assigned a cube map.
 * Use the wrap mode from the texture coordinate set at index 1.
 */
m_d3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX,
                        D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR | 1);
```

When you enable automatic texture coordinate generation, the system uses one of two formulas to compute the reflection vector for each vertex. When the D3DRS_LOCALVIEWER render state is set to TRUE, the formula used is the following:

$$R = 2(E \bullet N)N - E$$

In the preceding formula, *R* is the reflection vector being computed, *E* is the normalized position-to-eye vector, and *N* is the camera-space vertex normal. When the D3DRS_LOCALVIEWER render state is set to FALSE, the system uses the following formula.

$$R = 2N_z N - I$$

The *R* and *N* elements in this formula are identical to the previous formula. The N_z element is the world-space *z* of the vertex normal, and *I* is the vector (0,0,1) of an infinitely distant viewpoint. The system uses the reflection vector from either formula to select and address the appropriate face of the cube map.

Note

In most cases, applications should enable automatic normalization of vertex normals. To do this, set D3DRS_NORMALIZENORMALS to TRUE. If you do not enable this render state, the appearance of the environment map will be drastically different than you might expect.

[Visual Basic]

Texture coordinates that index a cubic-environment map aren't simple u, v style coordinates, as used when standard textures are applied. In fact, cubic environment maps don't use texture coordinates at all. In place of a set of texture coordinates, cubic environment maps require a 3-D vector. You must take care to specify a proper vertex format. In addition to telling the system how many sets of texture coordinates your application uses, you must provide information about how many elements are in each set. Microsoft® Direct3D® offers the **CONST_D3DFVFTEXTUREFORMATS** enumerated type which can be used to set up 3-D vectors. The members of this enumeration can be combined within a flexible vertex format description by using the **OR** operator.

```
'
' Create a flexible vertex format descriptor for a vertex that
' contains a position, normal, and one set of 3-D texture
' coordinates.
'/
Const D3DFVF CUSTOMVERTEX = (D3DFVF_XYZ Or D3DFVF_NORMAL Or _
                    D3DFVF_TEXCOORDSIZE3_0)
)
```

In some cases, such as diffuse light mapping, you use the camera-space vertex normal for the vector. In other cases, like specular environment mapping, you use a reflection vector. Because transformed vertex normals are widely understood, the information here concentrates on computing a reflection vector.

Computing a reflection vector on your own requires understanding of the position of each vertex, and of a vector from the viewpoint to that vertex. Direct3D can automatically compute the reflection vectors for your geometry. Using this feature

saves memory because you don't need to include texture coordinates for the environment map. It also reduces bandwidth and, in the case of a TnLHAL Device, it can be significantly faster than the computations your application can make on its own. To use this feature, in the texture stage that contains the cubic-environment map, set the D3DTSS_TEXCOORDINDEX texture stage state to a combination of the D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR flag and the index of a texture coordinate set. In some situations, like diffuse light mapping, you might use the D3DTSS_TCI_CAMERASPACENORMAL flag, which causes the system to use the transformed, camera-space, vertex normal as the addressing vector for the texture. The index is only used by the system to determine the wrapping mode for the texture. The following code example shows how this value is used.

```
'
' The m_D3DDevice variable is a valid Direct3DDevice8 object.
'
' Automatically generate texture coordinates for stage 2.
' This assumes that stage 2 is assigned a cube map.
' Use the wrap mode from the texture coordinate set at index 1.
'
m_D3DDevice.SetTextureStageState 2, D3DTSS_TEXCOORDINDEX, _
                    (D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR Or 1)
```

When you enable automatic texture coordinate generation, the system uses one of two formulas to compute the reflection vector for each vertex. When the D3DRS_LOCALVIEWER render state is set to TRUE, the formula used is the following:

$$R = 2(E \bullet N)N - E$$

In this formula, *R* is the reflection vector being computed, *E* is the normalized position-to-eye vector, and *N* is the camera-space vertex normal.
When the D3DRS_LOCALVIEWER render state is set to FALSE, the system uses the following formula.

$$R = 2N_z N - I$$

The *R* and *N* elements in this formula are identical to the previous formula. The N_z element is the world-space *z* of the vertex normal, and *I* is the vector (0,0,1) of an infinitely distant viewpoint. The system uses the reflection vector from either formula to select and address the appropriate face of the cube map.

Note

In most cases, applications should enable automatic normalization of vertex normals. To do this, set D3DRS_NORMALIZENORMALS to TRUE. If you do not enable this render state, the appearance of the environment map will be drastically different than you might expect.

Spherical Environment Mapping

Spherical environment mapping—often called sphere mapping—textures an object with an image or lighting effects of the space surrounding the object, where that space is encoded in a specially formatted texture map. The following topics introduce the concept of sphere mapping and provide information about how you can use it in your Microsoft® Direct3D® applications.

- What Are Spherical Environment Maps?
- Texture Coordinates for Spherical Environment Maps
- Applying Spherical Environment Maps

What Are Spherical Environment Maps?

Spherical environment maps, or sphere maps, are special textures that contain an image of the scene surrounding an object, or the lighting effects around the object. Unlike cubic environment maps, sphere maps don't directly represent an object's surroundings. The teapot image in the What Is Environment Mapping? topic shows the reflection effects you can achieve with sphere mapping.

[C++]
A sphere map is a 2-D representation of the full 360-degree view of the scene surrounding of an object, as if taken through a fish-eye lens. The following illustration is the sphere map used by the SphereMap Sample.

[Visual Basic]
A sphere map is a 2-D representation of the full 360-degree view of the scene surrounding of an object, as if taken through a fish-eye lens. The following illustration is the sphere map used by the C++ SphereMap Sample.

Texture Coordinates for Spherical Environment Maps

The texture coordinates that you specify for each vertex receiving an environment mapping should address the texture as a function of the reflective distortion created by the curvature of the surface. Applications must compute these texture coordinates for each vertex to achieve the desired effect. One simple and effective way to generate texture coordinates uses the vertex normal as input. Although several methods exist, the following formula is common among applications that perform environment mapping with sphere maps.

$$u = N_x/2 + 0.5$$
$$v = N_y/2 + 0.5$$

In these formulas, u and v are the texture coordinates being computed, and N_x and N_y are the x and y components of the camera-space vertex normal. The formula is simple but effective. If the normal has a positive x component, the normal points to the right, and the u coordinate is adjusted to address the texture appropriately. Likewise for the v coordinate: positive y indicates that the normal points up. The opposite is true for negative values in each component.
If the normal points directly at the camera, the resulting coordinates should receive no distortion. The +0.5 bias to both coordinates places the point of zero-distortion at the center of the sphere map, and a vertex normal of (0, 0, z) addresses this point. This formula doesn't account for the z component of the normal, but applications that use the formula can optimize computations by skipping vertices with a normal that has a positive z element. This works because, in camera space, if the normal points away from the camera (positive z), the vertex is culled when the object is rendered.

Applying Spherical Environment Maps

[C++]
You apply an environment map to objects in the same manner as for any other texture, by setting the texture to the appropriate texture stage with the **IDirect3DDevice8::SetTexture** method. Set the first parameter to the index for the desired texture stage, and set the second parameter to the address of the **IDirect3DCubeTexture8** interface returned when you created the cube-map texture for the environment map. You can set the color and alpha blending operations and arguments as needed to achieve the desired texture blending effects.

[Visual Basic]
You apply an environment map to objects in the same manner as for any other texture, by setting the texture to the appropriate texture stage with the **Direct3DDevice8.SetTexture** method. Set the first parameter to the index for the desired texture stage, and set the second parameter to the address of the **Direct3DCubeTexture8** object returned when you created the cube-map texture for the environment map. You can set the color and alpha blending operations and arguments as needed to achieve the desired texture blending effects.

Fog

The following topics introduce fog and present information about using various fog features in Microsoft® Direct3D® applications.

- Introduction to Fog
- Fog Formulas

- Fog Parameters
- Fog Blending
- Fog Color
- Pixel Fog
- Vertex Fog

Fog blending is controlled by render states; it is not part of the programmable pixel pipeline.

Introduction to Fog

Adding fog to a 3-D scene can enhance realism, provide ambiance or set a mood, and obscure artifacts sometimes caused when distant geometry comes into view. Microsoft® Direct3D® supports two fog models—pixel fog and vertex fog—each with its own features and programming interface.
Essentially, fog is implemented by blending the color of objects in a scene with a chosen fog color based on the depth of an object in a scene or its distance from the viewpoint. As objects grow more distant, their original color blends more and more with the chosen fog color, creating the illusion that the object is being increasingly obscured by tiny particles floating in the scene. The following illustration shows a scene rendered without fog, and a similar scene rendered with fog enabled.

In this illustration, the scene on the left has a clear horizon, beyond which no scenery is visible, even though it would be visible in the real world. The scene on the right obscures the horizon by using a fog color identical to the background color, making polygons appear to fade into the distance. By combining discrete fog effects with creative scene design you can add mood and soften the color of objects in a scene. Direct3D provides two ways to add fog to a scene, pixel fog and vertex fog, named for how the fog effects are applied. For details, see Pixel Fog and Vertex Fog. In short, pixel fog—also called table fog—is implemented in the device driver, and vertex fog is implemented in the Direct3D lighting engine.

Note

Regardless of which type of fog—pixel or vertex—you use, your application must provide a compliant projection matrix to ensure that fog effects are properly applied. This restriction applies even to applications that do not use the Direct3D transformation and lighting engine. For additional details about how you can provide an appropriate matrix, see A W-Friendly Projection Matrix.

Fog Formulas

[C++]
C++ applications can control how fog affects the color of objects in a scene by changing how Microsoft® Direct3D® computes fog effects over distance. The **D3DFOGMODE** enumerated type contains members that identify the three fog

formulas. All formulas calculate a fog factor as a function of distance, given parameters that your application sets. How distance is computed depends on the projection matrix or whether range-based fog is enabled. For more information, see Eye-Relative vs. Z-Based Depth and Range-Based Fog.

[Visual Basic]
Applications written in Microsoft® Visual Basic® can control how fog affects the color of objects in a scene by changing how Microsoft Direct3D® computes fog effects over distance. The **CONST_D3DFOGMODE** enumeration contains members that identify the three fog formulas. All formulas calculate a fog factor as a function of distance, given parameters that your application sets. How distance itself is computed depends on the projection matrix or whether range-based fog is enabled. For more information, see Eye-Relative vs. Z-Based Depth and Range-Based Fog.

D3DFOG_LINEAR

$$f = \frac{end - d}{end - start}$$

In this linear formula, *start* is the distance at which fog effects begin, *end* is the distance at which fog effects no longer increase, and *d* represents depth, or distance from the viewpoint, in a scene. Values for *d* increase as objects become more distant. Linear and exponential formulas are supported for both pixel fog and vertex fog.
D3DFOG_EXP

$$f = 1 / e^{d \times density}$$

D3DFOG_EXP2

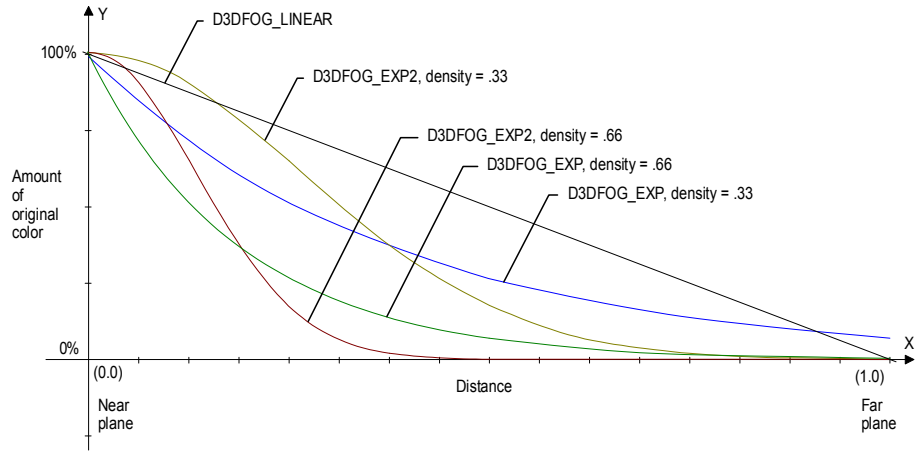
$$f = 1 / e^{(d \times density)^2}$$

In these two exponential formulas, *e* is the base of natural logarithms (approximately 2.71828), *density* is an arbitrary fog density that can range from 0.0 to 1.0, and *d* is depth, or distance from the viewpoint, in a scene.

Note

The system stores the fog factor in the alpha component of the specular color for a vertex. If your application performs its own transformation and lighting, you can insert fog factor values manually, to be applied by the system during rendering.

The following illustration graphs these formulas, using common values as in the formula parameters.



When Direct3D calculates fog effects, it uses the fog factor from one of the preceding equations in a blending formula, shown here.

$$C = f \cdot C_i + (1 - f) \cdot C_f$$

This formula effectively scales the color of the current polygon C_i by the fog factor f, and adds the product to the fog color C_f, scaled by the bitwise inverse of the fog factor. The resulting color value is a blend of the fog color and the original color, as a factor of distance. The formula applies to all devices supported in Microsoft DirectX® 7.0 and later. For the legacy ramp device, the fog factor scales the diffuse and specular color components, clamped to the range of 0.0 and 1.0, inclusive. The fog factor typically starts at 1.0 for the near plane and decreases to 0.0 at the far plane.

Fog Parameters

[C++]
Fog parameters are controlled through device render states. Both pixel and vertex fog types support all the fog formulas introduced in Fog Formulas. The **D3DFOGMODE** enumerated type defines constants that you can use to identify the fog formula you want Microsoft® Direct3D® to use. The D3DRS_FOGTABLEMODE render state controls the fog mode that Direct3D uses for pixel fog, and the D3DRS_FOGVERTEXMODE render state controls the mode for vertex fog.
When using the linear fog formula, you set the starting and ending distances through the D3DRS_FOGSTART and D3DRS_FOGEND render states. How the system interprets these values depends on the type of fog your application uses—pixel or vertex fog—and, when using pixel fog, if z-based or w-based depth is being used. For information z- and w-based pixel fog, see Eye-Relative vs. Z-Based Depth. The following table summarizes fog types and their start and end units.

Fog type	Fog start/end units
Pixel (Z)	Device space [0.0,1.0]
Pixel (W)	Camera space
Vertex	Camera space

The D3DRS_FOGDENSITY render state controls the fog density applied when an exponential fog formula is enabled. Fog density is essentially a weighting factor, ranging from 0.0 to 1.0 (inclusive), that scales the distance value in the exponent. The color that the system uses for fog blending is controlled through the D3DRS_FOGCOLOR device render state. For more information, see Fog Color and Fog Blending.

[Visual Basic]
Fog parameters are controlled through device render states. Both pixel and vertex fog types support all the fog formulas introduced in Fog Formulas. The **CONST_D3DFOGMODE** enumeration defines constants that you can use to identify the fog formula you want Microsoft® Direct3D® to use. The D3DRS_FOGTABLEMODE render state controls the fog mode that Direct3D uses for pixel fog, and the D3DRS_FOGVERTEXMODE render state controls the mode for vertex fog.
When using the linear fog formula, you set the starting and ending distances through the D3DRS_FOGSTART and D3DRS_FOGEND render states. How the system interprets these values depends on the type of fog your application uses—pixel or vertex fog—and, when using pixel fog, if z-based or w-based depth is being used. For information z- and w-based pixel fog, see Eye-Relative vs. Z-Based Depth. The following table summarizes fog types and their start and end units.

Fog Type	Fog Start/End Units
Pixel (Z)	Device space [0.0,1.0]
Pixel (W)	Camera space
Vertex	Camera space

The D3DRS_FOGDENSITY render state controls the fog density applied when an exponential fog formula is enabled. Fog density is essentially a weighting factor, ranging from 0.0 to 1.0 (inclusive), that scales the distance value in the exponent. The color that the system uses for fog blending is controlled through the D3DRS_FOGCOLOR device render state. Use the **D3DColorXRGB** function to generate valid values for this render state. For more information, see Fog Color and Fog Blending.

Fog Blending

[C++]

Fog blending refers to the application of the fog factor to the fog and object colors to produce the final color that appears in a scene, as discussed in Fog Formulas. The D3DRS_FOGENABLE render state controls fog blending. Set this render state to TRUE to enable fog blending as shown in the following example code. The default is FALSE.

```
//
// For this example, g_pDevice is a valid pointer
// to an IDirect3DDevice8 interface.
HRESULT hr;
hr = g_pDevice->SetRenderState(
                D3DRS_FOGENABLE,
                TRUE);
if FAILED(hr)
   return hr;
```

You must enable fog blending for both pixel fog and vertex fog. For information about using these types of fog, see Pixel Fog and Vertex Fog.

[Visual Basic]

Fog blending refers to the application of the fog factor to the fog and object colors to produce the final color that appears in a scene, as discussed in Fog Formulas. The D3DRS_FOGENABLE render state controls fog blending. Set this render state to True to enable fog blending as shown in the following example code. The default is False.

```
'
' For this example, d3dDevice contains a valid
' reference to a Direct3DDevice8 object.
'
On Local Error Resume Next

Call d3dDevice.SetRenderState( _
                D3DRS_FOGENABLE, _
                True)

If Err.Number <> D3D_OK Then
   ' Code to handle error goes here.
End If
```

You must enable fog blending for both pixel fog and vertex fog. For information about using these types of fog, see Pixel Fog and Vertex Fog.

Fog Color

[C++]

Fog color for both pixel and vertex fog is set through the D3DRS_FOGCOLOR render state. The render state values can be any RGB color, specified as an RGBA color; the alpha component is ignored.
The following C++ example sets the fog color to white.

```
/* For this example, the d3dDevice variable is
 * a valid pointer to an IDirect3DDevice8 interface.
 */
HRESULT hr;

hr = d3dDevice->SetRenderState(
            D3DRS_FOGCOLOR,
            0x00FFFFFF); // Highest 8 bits are not used.

if(FAILED(hr))
   return hr;
```

[Visual Basic]
Fog color for both pixel and vertex fog is set through the D3DRS_FOGCOLOR render state. The render state values can be any RGB color, as returned by the **D3DColorXRGB** function.
The following Microsoft® Visual Basic® example sets the fog color to white.

```
'
' For this example, the d3dDevice variable is
' a valid reference to a Direct3DDevice8 object.
'
On Local Error Resume Next

Call d3dDevice.SetRenderState( _
            D3DRS_FOGCOLOR, _
            D3DColorXRGB(1#, 1#, 1#))

If Err.Number <> D3D_OK Then
   ' Code to handle error goes here.
End If
```

Pixel Fog

This section introduces the concept of pixel fog and provides information about using it in Microsoft® Direct3D® applications. Information is divided into the following topics.

- About Pixel Fog
- Eye-Relative vs. Z-Based Depth
- Using Pixel Fog

About Pixel Fog

Pixel fog gets its name from the fact that it is calculated on a per-pixel basis in the device driver, unlike vertex fog, in which Microsoft® Direct3D® computes fog effects when it performs transformation and lighting. Pixel fog is sometimes called table fog because some drivers use a precalculated look-up table to determine the fog factor, using the depth of each pixel, to apply in blending computations.

[C++]
Pixel fog can be applied using any fog formula identified by members of the **D3DFOGMODE** enumerated type. Pixel-fog formula implementations are driver-specific, and if a driver doesn't support a complex fog formula, it should degrade to a less complex formula.

[Visual Basic]
Pixel fog can be applied using any fog formula identified by members of the **CONST_D3DFOGMODE** enumeration. Pixel-fog formula implementations are driver-specific, and if a driver doesn't support a complex fog formula, it should degrade to a less complex formula.

Note

As discussed in Range-Based Fog, pixel fog does not support range-based fog calculations.

Eye-Relative vs. Z-Based Depth

[C++]
To alleviate fog-related graphic artifacts caused by uneven distribution of z-values in a depth buffer, most hardware devices use eye-relative depth instead of z-based depth values for pixel fog. Eye-relative depth is essentially the w element from a homogeneous coordinate set. Microsoft® Direct3D® takes the reciprocal of the RHW element from a device space coordinate set to reproduce true w. If a device supports eye-relative fog, it sets the D3DPRASTERCAPS_WFOG flag in the **RasterCaps** member of the **D3DCAPS8** structure when you call the **IDirect3DDevice8::GetDeviceCaps** method.

Note

With the exception of the reference rasterizer, software devices always use z to calculate pixel fog effects.

When eye-relative fog is supported, the system automatically uses eye-relative depth rather than z-based depth if the provided projection matrix produces z-values in world space that are equivalent to w-values in device space. You set the projection matrix by calling the **IDirect3DDevice8::SetTransform** method, using the D3DTS_PROJECTION value and passing a **D3DMATRIX** structure that represents the desired matrix. If the projection matrix isn't compliant with this requirement, fog effects are not applied properly. For details about producing a compliant matrix, see

A W-Friendly Projection Matrix. The perspective projection matrix provided in What Is the Projection Transformation? produces a compliant projection matrix.
Direct3D uses the currently set projection matrix in its w-based depth calculations. As a result, an application must set a compliant projection matrix to receive the desired w-based features, even if it does not use the Direct3D transformation pipeline.
Direct3D checks the fourth column of the projection matrix. If the coefficients are [0,0,0,1] (for an affine projection) the system will use z-based depth values for fog. In this case, you must also specify the start and end distances for linear fog effects in device space, which ranges from 0.0 at the nearest point to the user, and 1.0 at the farthest point.

[Visual Basic]
To alleviate fog-related graphic artifacts caused by uneven distribution of z-values within a depth buffer, most hardware devices use eye-relative depth instead of z-based depth values for pixel fog. Eye-relative depth is essentially the w element from a homogeneous coordinate set. Microsoft® Direct3D® takes the reciprocal of the RHW element from a device space coordinate set to reproduce true w. If a device supports eye-relative fog, it sets the D3DPRASTERCAPS_WFOG flag in the **RasterCaps** member of the **D3DCAPS8** type when you call the **Direct3DDevice8.GetDeviceCaps** method.

Note

With the exception of the reference rasterizer, software devices always use z to calculate pixel fog effects.

When eye-relative fog is supported, the system automatically use eye-relative depth rather than z-based depth if the provided projection matrix produces z-values in world space that are equivalent to w-values in device space. You set the projection matrix by calling the **Direct3DDevice8.SetTransform** method, using the D3DTS_PROJECTION value from **CONST_D3DTRANSFORMSTATETYPE** and passing a **D3DMATRIX** type that represents the desired matrix. If the projection matrix isn't compliant with this requirement, fog effects are not applied properly. For details about producing a compliant matrix, see A W-Friendly Projection Matrix. The perspective projection matrix provided in What Is the Projection Transformation? produces a compliant projection matrix.
Direct3D uses the currently set projection matrix in its w-based depth calculations. As a result, an application must set a compliant projection matrix to receive the desired w-based features, even if it does not use the Direct3D transformation pipeline.
Direct3D checks the fourth column of the projection matrix, and if the coefficients are [0,0,0,1] (for an affine projection) the system will use z-based depth values for fog. In this case, you must also specify the start and end distances for linear fog effects in device space, which ranges from 0.0 at the nearest point to the user, and 1.0 at the farthest point.

Using Pixel Fog

Use the following steps to enable pixel fog in your application.

[C++]

Û To enable pixel fog in a C++ application

1. Enable fog blending by setting the D3DRS_FOGENABLE render state to TRUE.
2. Set the desired fog color in the D3DRS_FOGCOLOR render state.
3. Choose the fog formula to use by setting the D3DRS_FOGTABLEMODE render state to the corresponding member of the **D3DFOGMODE** enumerated type.
4. Set the fog parameters as desired for the selected fog mode in the associated render states. This includes the start and end distances for linear fog, and fog density for exponential fog mode.

The following example shows what these steps might look like in code.

```
// For brevity, error values in this example are not checked
// after each call. A real-world application should check
// these values appropriately.
//
// For the purposes of this example, g_pDevice is a valid
// pointer to an IDirect3DDevice8 interface.
void SetupPixelFog(DWORD Color, DWORD Mode)
{
   float Start = 0.5f,    // For linear mode
         End   = 0.8f,
         Density = 0.66;  // For exponential modes

   // Enable fog blending.
   g_pDevice->SetRenderState(D3DRS_FOGENABLE, TRUE);

   // Set the fog color.
   g_pDevice->SetRenderState(D3DRS_FOGCOLOR, Color);

   // Set fog parameters.
   if(D3DFOG_LINEAR == Mode)
   {
      g_pDevice->SetRenderState(D3DRS_FOGTABLEMODE, Mode);
      g_pDevice->SetRenderState(D3DRS_FOGSTART, *(DWORD *)(&Start));
      g_pDevice->SetRenderState(D3DRS_FOGEND,   *(DWORD *)(&End));
   }
   else
   {
      g_pDevice->SetRenderState(D3DRS_FOGTABLEMODE, Mode);
      g_pDevice->SetRenderState(D3DRS_FOGDENSITY, *(DWORD *)(&Density));
   }

}
```

Note

Some fog parameters are required as floating-point values, even though the **IDirect3DDevice8::SetRenderState** method only accepts **DWORD** values in the second parameter. The preceding example provides the floating-point values to **SetRenderState** without data translation by casting the addresses of the floating-point variables as **DWORD** pointers, and then dereferencing them.

[Visual Basic]

Û To enable pixel fog in a Visual Basic application

1. Enable fog blending by setting the D3DRS_FOGENABLE render state to True.
2. Set the desired fog color in the D3DRS_FOGCOLOR render state.
3. Choose the fog formula to use by setting the D3DRS_FOGTABLEMODE render state to the corresponding member of the **CONST_D3DFOGMODE** enumeration.
4. Set the fog parameters as desired for the selected fog mode in the associated render states. This includes the start and end distances for linear fog, and fog density for exponential fog mode.

The following example shows what these steps might look like in code.

```
' For brevity, error values in this example are not checked
' after each call. A real-world application should check
' these values appropriately.
'
' For the purposes of this example, d3dDevice is a valid
' reference to a Direct3DDevice8 object.
Sub SetupPixelFog(Color As Long, Mode As CONST_D3DFOGMODE)
    Dim StartFog As Single, _
        EndFog As Single, _
        Density As Single

    ' For linear mode
    StartFog = 0.5: EndFog = 0.8

    ' For exponential mode
    Density = 0.66

    ' Enable fog blending.
    Call d3dDevice.SetRenderState(D3DRS_FOGENABLE, True)

    ' Set the fog color.
    Call d3dDevice.SetRenderState(D3DRS_FOGCOLOR, Color)

    ' Set fog parameters.
    If Mode = D3DFOG_LINEAR Then
```

```
        Call d3dDevice.SetRenderState(D3DRS_FOGTABLEMODE, Mode)
        Call d3dDevice.SetRenderState(D3DRS_FOGSTART, StartFog)
        Call d3dDevice.SetRenderState(D3DRS_FOGEND, EndFog)
    Else
        Call d3dDevice.SetRenderState(D3DRS_FOGTABLEMODE, Mode)
        Call d3dDevice.SetRenderState(D3DRS_FOGDENSITY, Density)
    End If
End Sub
```

Vertex Fog

This section introduces vertex fog and provides details about using it in Microsoft® Direct3D® applications. Information is divided into the following topics.

- About Vertex Fog
- Range-Based Fog
- Using Vertex Fog

About Vertex Fog

When the system performs vertex fogging, it applies fog calculations at each vertex in a polygon, and then interpolates the results across the face of the polygon during rasterization. Vertex fog effects are computed by the Microsoft® Direct3D® lighting and transformation engine. For more information, see Fog Parameters.
If your application does not use Direct3D for transformation and lighting, it must perform fog calculations on its own. In this case, your application can place the fog factor that it computes in the alpha component of the specular color for each vertex. You are free to use whatever formulas you want—range-based, volumetric, or otherwise. Direct3D uses the supplied fog factor to interpolate across the face of each polygon. Applications that do not use Direct3D transformation and lighting need not set vertex fog parameters, but must still enable fog and set the fog color through the associated render states. For more information, see Fog Parameters.

Note

Applications that perform their own transformation and lighting must also perform their own vertex fog calculations. As a result, such an application need only enable fog blending and set the fog color through the associated render states, as described in Fog Blending and Fog Color.

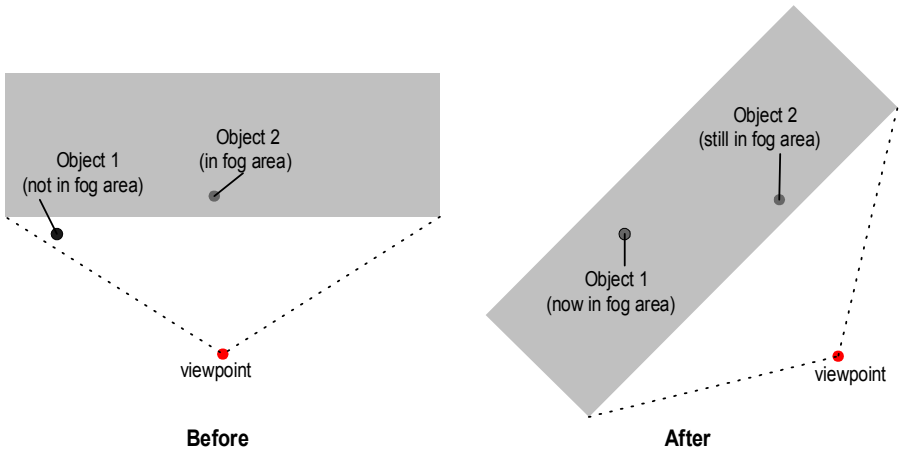
Range-Based Fog

Note

Microsoft® Direct3D® uses range-based fog calculations only when using vertex fog with the Direct3D transformation and lighting engine. This is because pixel fog is implemented in the device driver, and no hardware currently exists to support per-pixel range-based fog. If your application performs its own

transformation and lighting, it must perform its own fog calculations, range-based or otherwise.

Sometimes, using fog can introduce graphic artifacts that cause objects to be blended with the fog color in nonintuitive ways. For example, imagine a scene in which there are two visible objects: one distant enough to be affected by fog, and the other near enough to be unaffected. If the viewing area rotates in place, the apparent fog effects can change, even if the objects are stationary. The following illustration shows a top-down view of such a situation.



Range-based fog is another, more accurate, way to determine the fog effects. In range-based fog, Direct3D uses the actual distance from the viewpoint to a vertex for its fog calculations. Direct3D increases the effect of fog as the distance between the two points increases, rather than the depth of the vertex within in the scene, thereby avoiding rotational artifacts.

[C++]
If the current device supports range-based fog, it will set the D3DPRASTERCAPS_FOGRANGE capability flag in the **RasterCaps** member of the **D3DCAPS8** structure when you call the **IDirect3DDevice8::GetDeviceCaps** method. To enable range-based fog, set the D3DRS_RANGEFOGENABLE render state to TRUE.

[Visual Basic]
If the current device supports range-based fog, it will set the D3DPRASTERCAPS_FOGRANGE capability flag in the **RasterCaps** member of the **D3DCAPS8** type when you call the **Direct3DDevice8.GetDeviceCaps** method. To enable range-based fog, set the D3DRS_RANGEFOGENABLE render state to True.

Range-based fog is computed by Direct3D during transformation and lighting. As discussed in About Vertex Fog, applications that don't use the Direct3D transformation and lighting engine must also perform their own vertex fog calculations. In this case, provide the range-based fog factor in the alpha component of the specular component for each vertex.

Using Vertex Fog

Use the following steps to enable vertex fog in your application.

[C++]

To enable vertex fog in a C++ application

1. Enable fog blending by setting D3DRS_FOGENABLE to TRUE.
2. Set the fog color in the D3DRS_FOGCOLOR render state.
3. Choose the desired fog formula by setting the D3DRS_FOGVERTEXMODE render state to a member of the **D3DFOGMODE** enumerated type.
4. Set the fog parameters as desired for the selected fog formula in the render states.

The following example, written in C++, shows what these steps might look like in code.

```
// For brevity, error values in this example are not checked
// after each call. A real-world application should check
// these values appropriately.
//
// For the purposes of this example, g_pDevice is a valid
// pointer to an IDirect3DDevice8 interface.
void SetupVertexFog(DWORD Color, DWORD Mode, BOOL UseRange, FLOAT Density)
{
    float Start = 0.5f,    // Linear fog distances
          End   = 0.8f;

    // Enable fog blending.
    g_pDevice->SetRenderState(D3DRS_FOGENABLE, TRUE);

    // Set the fog color.
    g_pDevice->SetRenderState(D3DRS_FOGCOLOR, Color);

    // Set fog parameters.
    if(D3DFOG_LINEAR == Mode)
    {
        g_pDevice->SetRenderState(D3DRS_FOGVERTEXMODE, Mode);
        g_pDevice->SetRenderState(D3DRS_FOGSTART, *(DWORD *)(&Start));
        g_pDevice->SetRenderState(D3DRS_FOGEND,   *(DWORD *)(&End));
    }
    else
    {
        g_pDevice->SetRenderState(D3DRS_FOGVERTEXMODE, Mode);
```

```
    g_pDevice->SetRenderState(D3DRS_FOGDENSITY, *(DWORD *)(&Density));
  }

  // Enable range-based fog if desired (only supported for
  // vertex fog). For this example, it is assumed that UseRange
  // is set to a nonzero value only if the driver exposes the
  // D3DPRASTERCAPS_FOGRANGE capability.
  // Note: This is slightly more performance intensive
  //       than non-range-based fog.
  if(UseRange)
    g_pDevice->SetRenderState(
                D3DRS_RANGEFOGENABLE,
                TRUE);
}
```

Some fog parameters are required as floating-point values, even though the **IDirect3DDevice8::SetRenderState** method only accepts **DWORD** values in the second parameter. This example successfully provides the floating-point values to these methods without data translation by casting the addresses of the floating-point variables as **DWORD** pointers, and then dereferencing them.

[Visual Basic]

To enable vertex fog from a Visual Basic application

1. Enable fog blending by setting D3DRS_FOGENABLE to True.
2. Set the fog color in the D3DRS_FOGCOLOR render state.
3. Choose the desired fog formula by setting the D3DRS_FOGVERTEXMODE render state to a member of the **CONST_D3DFOGMODE** enumeration.
4. Set the fog parameters as desired for the selected fog formula in the render states.

The following Microsoft® Visual Basic® example code shows what these steps might look like.

```
' For brevity, error values in this example are not checked
' after each call. A real-world application should check
' these values appropriately.
'
' For the purposes of this example, d3dDevice is a valid
' reference to a Direct3DDevice8 object.
Sub SetupVertexFog(Color As Long, Mode As CONST_D3DFOGMODE, _
            UseRange As Boolean, Optional Density As Single)

  Dim StartFog As Single, _
    EndFog As Single

  ' Set linear fog distances
```

```
    StartFog = 0.5: EndFog = 0.8

    ' Enable fog blending.
    Call d3dDevice.SetRenderState(D3DRS_FOGENABLE, True)

    ' Set the fog color.
    Call d3dDevice.SetRenderState(D3DRS_FOGCOLOR, Color)

    ' Set fog parameters.
    If Mode = D3DFOG_LINEAR Then
        Call d3dDevice.SetRenderState(D3DRS_FOGVERTEXMODE, Mode)
        Call d3dDevice.SetRenderStateSingle(D3DRS_FOGSTART, StartFog)
        Call d3dDevice.SetRenderStateSingle(D3DRS_FOGEND, EndFog)
    Else
        Call d3dDevice.SetRenderState(D3DRS_FOGVERTEXMODE, Mode)
        Call d3dDevice.SetRenderStateSingle(D3DRS_FOGDENSITY, Density)
    End If

    ' Enable range-based fog if desired (only supported for vertex
    ' fog). For this example, it is assumed that UseRange is set to
    ' True only if the driver exposes the D3DPRASTERCAPS_FOGRANGE
    ' capability.
    ' Note: This is slightly more performance intensive
    '       than non-range-based fog.
    If UseRange = True Then
        Call d3dDevice.SetRenderState( _
                    D3DRS_RANGEFOGENABLE, True)
    End If
End Sub
```

Geometry Blending

This section contains information about geometry blending. The following topics are discussed.

- About Geometry Blending
- Blending Transform and Render States
- Blending Weights
- Using Geometry Blending

About Geometry Blending

Microsoft® Direct3D® enables an application to increase the realism of its scenes by rendering segmented polygonal objects—especially characters—that have smoothly

blended joints. These effects are often referred to as *skinning*. The system achieves this effect by applying additional world transformation matrices to a single set of vertices to create multiple results, and then performing a linear blend between the resultant vertices to create a single set of geometry for rendering. The following image of a banana illustrates this.

This image shows how you might imagine the geometry-blending process. In a single rendering call, the system takes the vertices for the banana, transforms them twice—once without modification, and once with a simple rotation—and blends the results to create a bent banana. The system blends the vertex position, as well as the vertex normal when lighting is enabled. Applications are not limited to two blending paths; Direct3D can blend geometry between as many as four world matrices, including the standard world matrix, D3DTS_WORLD.

[C++]

Note

When lighting is enabled, vertex normals are transformed by a corresponding inverse world-view matrix, weighted in the same way as the vertex position computations. The system normalizes the resulting normal vector if the D3DRS_NORMALIZENORMALS render state is set to TRUE.

Without geometry blending, dynamic articulated models are often rendered in segments. For instance, consider a 3-D model of the human arm. In the simplest view, an arm has two parts: the upper arm which connects to the body, and the lower arm, which connects to the hand. The two are connected at the elbow, and the lower arm rotates at that point. An application that renders an arm might retain vertex data for the upper and lower arm, each with a separate world transformation matrix. The following code example illustrates this.

```
typedef struct _Arm {
    VERTEX upper_arm_verts[200];
    D3DMATRIX matWorld_Upper;

    VERTEX lower_arm_verts[200];
    D3DMATRIX matWorld_Lower;
} ARM, *LPARM;

ARM MyArm; // This needs to be initialized.
```

To render the arm, two rendering calls are made, as shown in the following code.

```
// Render the upper arm.
d3dDevice->SetTransform( D3DTS_WORLD, &MyArm.matWorld_Upper );
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, numFaces );

// Render the lower arm, updating its world matrix to articulate
// the arm by pi/4 radians (45 degrees) at the elbow.
```

```
MyArm.matWorld_Lower = RotateMyArm(MyArm.matWorld, pi/4);
d3dDevice->SetTransform( D3DTS_WORLD, &MyArm.matWorld_Lower );
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, numFaces );
```

[Visual Basic]

Note

When lighting is enabled, vertex normals are transformed by a corresponding inverse world-view matrix, weighted in the same way as the vertex position computations. The system normalizes the resulting normal vector if the D3DRS_NORMALIZENORMALS render state is set to True.

Without geometry blending, dynamic articulated models are often rendered in segments. For instance, consider a 3-D model of the human arm. In the simplest view, an arm has two parts: the upper arm which connects to the body, and the lower arm, which connects to the hand. The two are connected at the elbow, and the lower arm rotates at that point. An application that renders an arm might retain vertex data for the upper and lower arm, each with a separate world transformation matrix. The following code example illustrates this.

```
Type Arm
   upper_arm_verts(200) As D3DVERTEX
   matWorld_Upper As D3DMATRIX

   lower_arm_verts(200) As D3DVERTEX
   matWorld_Lower As D3DMATRIX
End Type

Dim MyArm As Arm ' This needs to be initialized.
```

To render the arm, two rendering calls are made, as shown in the following code.

```
' Render the upper arm.
Call d3DDevice.SetTransform(D3DTS_WORLD, MyArm.matWorld_Upper)
Call d3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, 0, numFaces)

' Render the lower arm, updating its world matrix to articulate
' the arm by pi/4 radians (45 degrees) at the elbow.
MyArm.matWorld_Lower = RotateMyArm(MyArm.matWorld, pi / 4)
Call d3DDevice.SetTransform(D3DTS_WORLD, MyArm.matWorld_Lower)
Call d3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, 0, numFaces)
```

The following image is a banana, modified to use this technique.

The differences between the blended geometry and the nonblended geometry are obvious. This example is somewhat extreme. In a real-world application, the joints of segmented models are designed so that seams are not as obvious. However, seams are visible at times, which presents constant challenges for model designers.
Geometry blending in Direct3D presents an alternative to the classic segmented-modeling scenario. However, the improved visual quality of segmented objects comes at the cost of the blending computations during rendering. To minimize the impact of these additional operations, the Direct3D geometry pipeline is optimized to blend geometry with the least possible overhead. Applications that intelligently use the geometry blending services offered by Direct3D can improve the realism of their characters while avoiding serious performance repercussions.

Blending Transform and Render States

[C++]
The **IDirect3DDevice8::SetTransform** method recognizes the **D3DTS_WORLD** and **D3DTS_WORLD*n*** macros, which correspond to values that can be defined by the **D3DTS_WORLDMATRIX** macro. These macros are used to identify the matrices between which geometry will be blended.
The **D3DRENDERSTATETYPE** enumerated type includes the D3DRS_VERTEXBLEND render state to enable and control geometry blending. Valid values for this render state are defined by the **D3DVERTEXBLENDFLAGS** enumerated type. If geometry blending is enabled, the vertex format must include the appropriate number of blending weights.

[Visual Basic]
The **Direct3DDevice8.SetTransform** method recognizes the D3DTS_WORLD, D3DTS_WORLD1, D3DTS_WORLD2, and D3DTS_WORLD3 members of the **CONST_D3DTRANSFORMSTATETYPE** enumeration to identify the matrices between which geometry will be blended. The D3DTS_WORLD member might be thought of as D3DTS_WORLD0.
The **CONST_D3DRENDERSTATETYPE** enumeration includes the D3DRS_VERTEXBLEND render state to enable and control geometry blending. Valid values for this render state are defined by the **CONST_D3DVERTEXBLENDFLAGS** enumerated type. If geometry blending is enabled, the vertex format must include the appropriate number of blending weights.

See Also

Setting Blending Matrices, Enabling Geometry Blending, Blending Weights

Blending Weights

A blending weight, sometimes called a beta weight, controls the extent to which a given world matrix affects a vertex. Blending weights are floating-point values that range from 0.0 to 1.0, encoded in the vertex format, where a value of 0.0 means the

vertex is not blended with that matrix, and 1.0 means that the vertex is affected in full by the matrix.

[C++]
Geometry blending weights are encoded in the vertex format, appearing immediately after the position for each vertex, as described in About Vertex Formats. You communicate the number of blending weights in the vertex format by including one of the D3DFVF_XYZB1 through D3DFVF_XYZB5 flexible vertex format flags in the vertex description that you provide to the Microsoft® Direct3D® rendering methods.

[Visual Basic]
Geometry blending weights are encoded in the vertex format, appearing immediately after the position for each vertex, as described in About Vertex Formats. You communicate the number of blending weights in the vertex format by including one of the D3DFVF_XYZB1 through D3DFVF_XYZB5 flexible vertex format flags in the vertex description that you provide to the Microsoft® Direct3D® rendering methods.

The system performs a linear blend between the weighted results of the blend matrices. The following is the complete blending formula.

$$vBlend = V_1 W_1 + \ldots + V_{n-1} W_{n-1} + V_n \left(1.0 - \sum_{i=1}^{n} W_i\right)$$

[C++]
In the preceding formula, *vBlend* is the output vertex, the v-elements are the vertices produced by the applied world matrix (**D3DTS_WORLD*n***). The W elements are the corresponding weight values within the vertex format. A vertex blended between *n* matrices can have *n-1* blending weight values, one for each blending matrix, except the last. The system automatically generates the weight for the last world matrix so that the sum of all weights is 1.0, expressed in sigma notation here. This formula can be simplified for each of the cases supported by Direct3D.

[Visual Basic]
In the preceding formula, *vBlend* is the output vertex, the v-elements are the vertices produced by the applied world matrix. The W elements are the corresponding weight values within the vertex format. A vertex blended between *n* matrices can have *n-1* blending weight values, one for each blending matrix, except the last. The system automatically generates the weight for the last world matrix so that the sum of all weights is 1.0, expressed in sigma notation here. This formula can be simplified for each of the cases supported by Direct3D.

$$vBlend = V_1W_1 + V_2(1.0 - W_1)$$

$$vBlend = V_1W_1 + V_2W_2 + V_3(1.0 - (W_1+W_2))$$

$$vBlend = V_1W_1 + V_2W_2 + V_3W_3 + V_4(1.0 - (W_1+W_2+W_3))$$

These are the simplified forms of the complete blending formula for the two, three, and four blend matrix cases.

Note

Although Direct3D includes flexible vertex format descriptors to define vertices that contain up to five blending weights, only three can be used in this release of Microsoft DirectX®.

Using Geometry Blending

This section illustrates how you use geometry blending in your application. The following topics are discussed.

- Defining Vertices for Blending
- Setting Blending Matrices
- Enabling Geometry Blending

Defining Vertices for Blending

[C++]

The following user-defined structure can be used for vertices that will be blended between two matrices.

```
//
// The flexible vertex format descriptor for this vertex would be:
//
//    FVF = D3DFVF_XYZB1 | D3DFVF_NORMAL | D3DFVF_TEX1;
//
struct D3DBLENDVERTEX {
    D3DVECTOR v;
    FLOAT     blend;
    D3DVECTOR n;
    FLOAT     tu, tv;
};
```

As described in Blending Weights, the blend weight must appear after the position and RHW data in the flexible vertex format, and before the vertex normal.
Notice that the preceding vertex format contains only one blending weight value. This is because there will be two world matrices, defined in the **D3DTS_WORLDMATRIX**(0) and **D3DTS_WORLDMATRIX**(1) transform

states. The system blends each vertex between these two matrices using the single weight value. For three matrices, only two weights are required, and so on.

[Visual Basic]
The following user-defined type can be used for vertices that will be blended between two matrices.

```
'
' The flexible vertex format descriptor for this vertex would be:
'
'    FVF = (D3DFVF_XYZB1 Or D3DFVF_NORMAL Or D3DFVF_TEX1)
'
Type D3DBLENDVERTEX
   v As D3DVECTOR
   blend As Single
   n As D3DVECTOR
   tu As Single
   tv As Single
End Type
```

As described in Blending Weights, the blend weight must appear after the position and RHW data in the flexible vertex format, and before the vertex normal.
Notice that the preceding vertex format contains only one blending weight value. This is because there will be two world matrices, defined in the D3DTS_WORLD and D3DTS_WORLD1 transform states. The system blends each vertex between these two matrices using the single weight value. For three matrices, only two weights are required, and so on.

Note

Defining skin weights is easy. Using a linear function of the distance between joints is good start, but a smoother sigmoid function looks better. Choosing a skin-weight distribution function can result in sharp creases at joints, if desired, by using a large variation in skin weight over a short distance.

See Also

[C++]
About Vertex Formats, Flexible Vertex Format Flags

[Visual Basic]
About Vertex Formats, Flexible Vertex Format Flags

Setting Blending Matrices

[C++]
You set the transformation matrices between which the system blends by calling the **IDirect3DDevice8::SetTransform** method. Set the first parameter to a value defined by the **D3DTS_WORLDMATRIX** macro, and set the second parameter to the address of the matrix to be set.
The following C++ code example sets two world matrices, between which geometry is blended to create the illusion of a jointed arm.

```
// For this example, the d3dDevice variable is assumed to be a
// valid pointer to an IDirect3DDevice8 interface for an initialized
// 3-D scene.
float     BendAngle = 3.1415926f / 4.0f; // 45 degrees
D3DMATRIX matUpperArm, matLowerArm;

// The upper arm is immobile. Use the identity matrix.
D3DXMatrixIdentity( matUpperArm );
d3dDevice->SetTransform( D3DTS_WORLDMATRIX(0), &matUpperArm );

// The lower arm rotates about the x-axis, attached to the upper arm.
D3DXMatrixRotationX( matLowerArm, -BendAngle );
d3dDevice->SetTransform( D3DTS_WORLDMATRIX(1), &matLowerArm );
```

Setting a blending matrix merely causes the system to cache the matrix for later use; it doesn't instruct the system to begin blending vertices. For more information, see Enabling Geometry Blending.

[Visual Basic]
You set the transformation matrices between which the system blends by calling the **Direct3DDevice8.SetTransform** method. Set the first parameter to a D3DTS_WORLD member from the **CONST_D3DTRANSFORMSTATETYPE** enumeration, and set the second parameter to the address of the matrix to be set.
The following Microsoft® Visual Basic® code example sets two world matrices between which geometry will be blended to create the illusion of a jointed arm.

```
' For this example, the d3dDevice variable is assumed to be a valid
' reference to a Direct3DDevice8 object for an initialized 3-D scene.
' The dx variable is a valid reference to a DirectX7 object.
Dim BendAngle As Single
Dim matUpperArm As D3DMATRIX, _
    matLowerArm As D3DMATRIX

BendAngle = 3.1415926 / 4#  ' 45 degrees

' The upper arm is immobile. Use the identity matrix.
D3DXMatrixIdentity matUpperArm
Call d3dDevice.SetTransform(D3DTS_WORLD, matUpperArm)
```

```
' The lower arm rotates about the x-axis, attached to the upper arm.
D3DXMatrixRotationX matLowerArm, -nBendAngle
Call d3dDevice.SetTransform(D3DTS_WORLD1, matLowerArm)
```

Setting a blending matrix merely causes the system to cache the matrix for later use; it does not instruct the system to begin blending vertices. For more information, see Enabling Geometry Blending.

Enabling Geometry Blending

[C++]
Geometry blending is disabled by default. To enable geometry blending, call the **IDirect3DDevice8::SetRenderState** method to set the D3DRS_VERTEXBLEND render state to a value from the **D3DVERTEXBLENDFLAGS** enumerated type. The following code example shows what this call might look like when setting the render state for a blend between two world matrices.

```
d3dDevice->SetRenderState( D3DRS_VERTEXBLEND, D3DVBF_1WEIGHTS );
```

When D3DRS_VERTEXBLEND is set to any value other than D3DVBF_DISABLE, the system assumes that the appropriate number of blending weights will be included in the vertex format. It is your responsibility to provide a compliant vertex format, and to provide a proper description of that format to the Microsoft® Direct3D® rendering methods. For more information, see Defining Vertices for Blending.
When enabled, the system performs geometry blending for all objects rendered by the DrawPrimitive rendering methods.

[Visual Basic]
Geometry blending is disabled by default. To enable geometry blending, call the **Direct3DDevice8.SetRenderState** method to set the D3DRS_VERTEXBLEND render state to a value from the **CONST_D3DVERTEXBLENDFLAGS** enumeration. The following code example shows what this call might look like when setting the render state for a blend between two world matrices.

```
Call d3dDevice.SetRenderState(D3DRS_VERTEXBLEND, D3DVBF_1WEIGHT)
```

When D3DRS_VERTEXBLEND is set to any value other than D3DVBF_DISABLE, the system assumes that the appropriate number of blending weights will be included in the vertex format. It is your responsibility to provide a compliant vertex format, and to provide a proper description of that format to the Microsoft® Direct3D® rendering methods. For more information, see Defining Vertices for Blending.
When enabled, the system performs geometry blending for all objects rendered by the DrawPrimitive rendering methods.

Indexed Vertex Blending

This section contains information about indexed vertex blending for the Microsoft® Direct3D® fixed function vertex pipeline. The following topics are discussed.

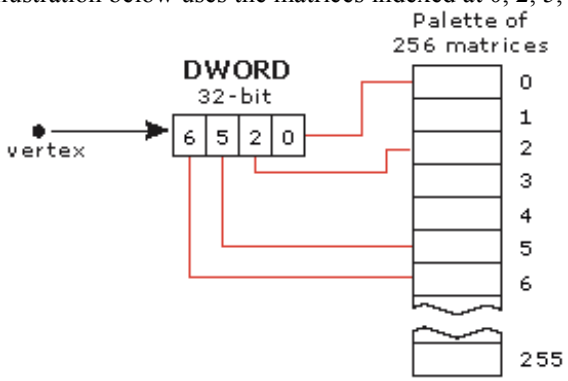
- About Indexed Vertex Blending
- Indexed Vertex Blending Transform and Render States
- Using Indexed Vertex Blending

About Indexed Vertex Blending

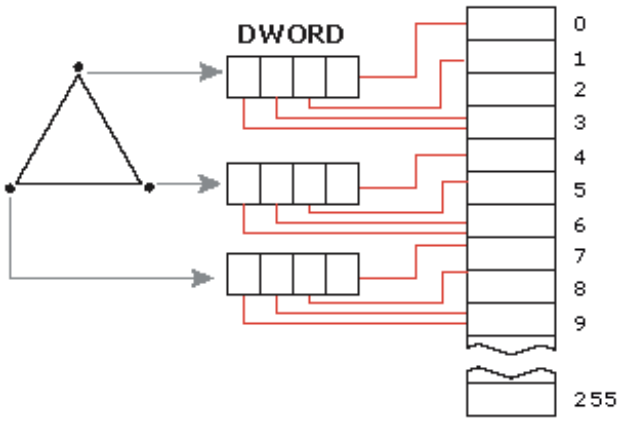
Indexed vertex blending extends the vertex blending support in Microsoft® Direct3D® to allow matrices to be used for blending. These matrices are referred to by using a matrix index. These indices are supplied on a per-vertex basis and refer to a palette of up to 256 matrices. Each index is 8 bits and each vertex can have up to four indices, which allows four matrices to be blended per vertex. The indices are packed into a **DWORD**. Because indices are specified on a per-vertex basis, up to 12 matrices can affect a single triangle, and any matrix in the palette can affect the vertices of one draw call. This approach has the following advantages.

- It enables more matrices to affect a single triangle.
- It enables more blended triangles to be passed in the same draw call.
- It makes vertex blending independent of triangle indices. This enables progressive meshes to work in conjunction with vertex blending.

One disadvantage of this approach is that it does not work with curved-surface primitives when tessellation occurs before vertex processing.
The following illustration demonstrates how four matrices can affect a vertex. Each vertex has up to four indices, so four matrices can be blended per vertex. The illustration below uses the matrices indexed at 0, 2, 5, and 6.



The illustration below demonstrates how up to 12 matrices can affect a triangle.
Using indices specified on a per-vertex basis, up to 12 matrices can affect the triangle.



The following formula determines the general case for how matrices effect a vertex.

V_{model} is the input model space vertex position. *Index0..Index3* are the per-vertex matrix indices packed into a **DWORD**. *M[]* is the array of world matrices that get indexed into. $b_0..b_2$ are the blend weights. V_{world} is the output world space vertex position.

Indexed Vertex Blending Transform and Render States

[C++]
Transform states 256-511 are reserved to store up to 256 matrices that can be indexed using 8-bit indices. Use the macro **D3DTS_WORLDMATRIX** to map indices 0-255 to the corresponding transform states. The following code example shows how to use the **IDirect3DDevice8::SetTransform** method to set the matrix at transform state number 256 to an identity matrix.

```
D3DMATRIX matBlend1;

D3DXMatrixIdentity( &matBlend1 );
d3dDevice->SetTransform( D3DTS_WORLDMATRIX(0), &matBlend );
```

To enable or disable indexed vertex blending, set the **D3DRS_INDEXEDVERTEXBLENDENABLE** render state to TRUE. When the render state is enabled ,you must pass matrix indices as packed **DWORD**s with every vertex. When this render state is disabled and vertex blending is enabled, it is equivalent to having the matrix indices 0, 1, 2, and 3 in every vertex. The code example below uses the **IDirect3DDevice8::SetRenderState** method to enable indexed vertex blending.

```
d3dDevice->SetRenderState( D3DRS_INDEXEDVERTEXBLENDENABLE, TRUE );
```

To enable or disable vertex blending, set the **D3DRS_VERTEXBLEND** render state to a value other than D3DRS_DISABLE from the **D3DVERTEXBLENDFLAGS** enumerated type. If this render state is not set to D3DRS_DISABLE, then you must pass the required number of weights for each vertex. The following code example uses **SetRenderState** to enable vertex blending with three weights for each vertex.

```
d3dDevice->SetRenderState( D3DRS_VERTEXBLEND, D3DVBF_3WEIGHTS );
```

[Visual Basic]
Transform states 256-511 are reserved to store up to 256 matrices that can be indexed using 8-bit indices. Use the function below to map indices 0-255 to the corresponding transform states.

```
Function D3DTS_WORLDMATRIX(index as Long) As Long
   D3DTS_WORLDMATRIX=(index + 256)
End Function
```

The following code example shows how to use the **Direct3DDevice8.SetTransform** method to set the matrix at transform state number 256 to an identity matrix.

```
Dim matBlend1 As D3DMATRIX

Call D3DXMatrixIdentity(matBlend1)
Call m_D3DDevice.SetTransform(D3DTS_WORLDMATRIX(0), matBlend)
```

To enable or disable indexed vertex blending, set the **D3DRS_INDEXVERTEXBLENDENABLE** render state to True. When the render state is enabled, you must pass matrix indices as packed **Long**s with every vertex. When this render state is disabled and vertex blending is enabled, it is equivalent to having the matrix indices 0, 1, 2, and 3 in every vertex. The code example below uses the **Direct3DDevice8.SetRenderState** method to enable indexed vertex blending.

```
Call m_D3DDevice.SetRenderState(D3DRS_INDEXVERTEXBLENDENABLE, True)
```

To enable or disable vertex blending, set the **D3DRS_VERTEXBLEND** render state to a value other than D3DRS_DISABLE from the **D3DVERTEXBLENDFLAGS** enumerated type. If this render state is not set to D3DRS_DISABLE, then you must pass the required number of weights for each vertex. The following code example uses **SetRenderState** to enable vertex blending with three weights for each vertex.

```
Call m_D3DDevice.SetRenderState(D3DRS_VERTEXBLEND, D3DVBF_3WEIGHTS)
```

Using Indexed Vertex Blending

The following topics are discussed in this section.

- Determining Support for Indexed Vertex Blending
- Passing Matrix Indices to Direct3D

Determining Support for Indexed Vertex Blending

[C++]
To determine the maximum size for the indexed vertex blending matrix, check the **MaxVertexBlendMatrix** member of the **D3DCAPS8** structure. The code example below uses the **IDirect3DDevice8::GetDeviceCaps** method to retrieve this size.

```
D3DCAPS8 d3dCaps;

d3dDevice->GetDeviceCaps( &d3dCaps );
IndexedMatrixMaxSize = d3dCaps.MaxVertexBlendMatrixIndex;
```

If the value set in **MaxVertexBlendMatrix** is 0, then the device does not support indexed matrices.

[Visual Basic]
To determine the maximum size for the indexed vertex blending matrix, check the **MaxVertexBlendMatrix** member of the **D3DCAPS8** structure. The code example below uses the **Direct3DDevice8.GetDeviceCaps** method to retrieve this size.

```
Dim d3dCaps As D3DCAPS8

Call D3DDevice.GetDeviceCaps(d3dCaps)
IndexedMatrixMaxSize = d3dCaps.MaxVertexBlendMatrixIndex
```

If the value set in **MaxVertexBlendMatrix** is 0, then the device does not support indexed matrices.

Note

When software vertex processing is used, 256 matrices can be used for indexed vertex blending, with or without normal blending.

Passing Matrix Indices to Direct3D

[C++]
World matrix indices can be passed to Microsoft® Direct3D® by using legacy vertex shaders (FVF) or declarations.
The code example below shows how to use legacy vertex shaders.

```
struct VERTEX
{
    float x,y,z;
    float weight;
    DWORD matrixIndices;
    float normal[3];
};

#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZB2 | D3DFVF_LASTBETA_UBYTE4 |
```

```
                D3DFVF_NORMAL);
```

When a legacy vertex shader is used, matrix indices are passed together with vertex positions using **D3DFVF_XYZBn** flags. Matrix indices are passed as bytes inside a **DWORD** and must be present immediately after the last vertex weight. Vertex weights are also passed using **D3DFVF_XYZBn**. A packed **DWORD** contains index3, index2, index1, and index0, where index0 is located in the lowest byte of the **DWORD**. The number of used world-matrix indices is equal to the number passed to the number of matrices used for blending as defined by **D3DRS_VERTEXBLEND**. When a declaration is used, **D3DVSDE_BLENDINDICES** defines the input vertex register to get matrix indices from. Matrix indices must be passed as **D3DVSDT_PACKEDBYTE**.
The code example below shows how to use declarations. Note that the order of the components is no longer important unless using a fixed-function vertex shader.

```
struct VERTEX
{
   float x,y,z;
   float weight;
   DWORD matrixIndices;
   float normal[3];
}

DWORD decl[] =
{
   D3DVSD_STREAM(0),
   D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3),
   D3DVSD_REG(D3DVSDE_BLENDWEIGHT, D3DVSDT_FLOAT1),
   D3DVSD_REG(D3DVSDE_BLENDINDICES, D3DVSDT_UBYTE),
   D3DVSD_REG(D3DVSDE_NORMAL, D3DVSDT_FLOAT3),
   D3DVSD_END()
};
```

[Visual Basic]
World matrix indices can be passed to Microsoft® Direct3D® by using legacy vertex shaders (FVF) or declarations.
The code example below shows how to use legacy vertex shaders.

```
Private Type VERTEX
   x As Single
   y As Single
   z As Single
   weight As Single
   matrixIndices As Long
   normal(2) As Single
End Type
```

```
Const D3DFVF_CUSTOMVERTEX = (D3DFVF_XYZB2 Or D3DFVF_NORMAL Or _
                    D3DFVF_LASTBETA_UBYTE4)
```

When a legacy vertex shader is used, matrix indices are passed together with vertex positions using **D3DFVF_XYZBn** flags. Matrix indices are passed as a packed byte and must be present immediately after the last vertex weight. Vertex weights are also passed using **D3DFVF_XYZBn**. A packed byte contains index3, index2, index1, and index0, where index0 is located in the lowest byte of the byte. The number of used world-matrix indices is equal to the number passed to the number of matrices used for blending as defined by **D3DRS_VERTEXBLEND**.
When a declaration is used, **D3DVSDE_BLENDINDICES** defines the input vertex register to get matrix indices from. Matrix indices must be passed as **D3DVSDT_PACKEDBYTE**.
The code example below shows how to use declarations. Note that the order of the components is no longer important.

```
Private Type VERTEX
   x As Single
   y As Single
   z As Single
   matrixIndices As Long
   weight As Single
   normal(2) As Single
End Type

Dim decl(5) As Long

decl(0) = D3DVSD_STREAM(0)
decl(1) = D3DVSD_REG(D3DVSDE_POSITION, D3DVSDT_FLOAT3)
decl(2) = D3DVSD_REG(D3DVSDE_BLENDINDICES, D3DVSDT_PACKEDBYTE)
decl(3) = D3DVSD_REG(D3DVSDE_BLENDWEIGHT, D3DVSDT_FLOAT1)
decl(4) = D3DVSD_REG(D3DVSDE_NORMAL, D3DVSDT_FLOAT3)
decl(5) = D3DVSD_END()
```

Stencil Buffer Techniques

Applications use the stencil buffer to mask pixels in an image. The mask controls whether the pixel is drawn. For more information on the stencil buffer, see Stencil Buffers.
Microsoft® Direct3D® applications can achieve a wide range of special effects with the stencil buffer. Some of the more common effects are discussed in this section.

- Dissolves, Fades, and Swipes
- Decaling

- Compositing
- Outlines and Silhouettes

Dissolves, Fades, and Swipes

Increasingly, applications employ special effects that are commonly used in movies and video, such as dissolves, swipes, and fades.

In a dissolve, one image is gradually replaced by another in a smooth sequence of frames. Although Microsoft® Direct3D® provides methods of using multiple texture blending to achieve the same effect, applications that use the stencil buffer for dissolves can use texture-blending capabilities for other effects while they do a dissolve.

When your application performs a dissolve, it must render two different images. It uses the stencil buffer to control which pixels from each image are drawn to the rendering target surface. You can define a series of stencil masks and copy them into the stencil buffer on successive frames. Alternately, you can define a base stencil mask for the first frame and alter it incrementally.

At the beginning of the dissolve, your application sets the stencil function and stencil mask so that most of the pixels from the starting image pass the stencil test. Most of the pixels from the ending image should fail the stencil test. On successive frames, the stencil mask is updated so that fewer and fewer of the pixels in the starting image pass the test. As the frames progress, fewer and fewer of the pixels in the ending image fail the test. In this manner, your application can perform a dissolve using any arbitrary dissolve pattern.

Fading in or fading out is a special case of dissolving. When fading in, the stencil buffer is used to dissolve from a black or white image to a rendering of a 3-D scene. Fading out is the opposite, your application starts with a rendering of a 3-D scene and dissolves to black or white. The fade can be done using any arbitrary pattern you want to employ.

Direct3D applications use a similar technique for swipes. For example, when an application performs a left-to-right swipe, the ending image appears to slide gradually on top of the starting image from left to right. As in a dissolve, you must define a series of stencil masks that are loaded into the stencil buffer on successive frames, or successively modify the starting stencil mask. The stencil masks are used to disable the writing of pixels from the starting image and to enable the writing of pixels from the ending image.

A swipe is somewhat more complex than a dissolve in that your application must read pixels from the ending image in the reverse order of the swipe. That is, if the swipe is moving from left to right, your application must read pixels from the ending image from right to left.

Decaling

Microsoft® Direct3D® applications use decaling to control which pixels from a particular primitive image are drawn to the rendering target surface. Applications apply decals to the images of primitives to enable coplanar polygons to render correctly.

For instance, when applying tire marks and yellow lines to a roadway, the markings should appear directly on top of the road. However, the z values of the markings and the road are the same. Therefore, the depth buffer might not produce a clean separation between the two. Some pixels in the back primitive may be rendered on top of the front primitive and vice versa. The resulting image appears to shimmer from frame to frame. This effect is called *Z Fighting* or *flimmering*.
To solve this problem, use a stencil to mask the section of the back primitive where the decal will appear. Turn off z-buffering and render the image of the front primitive into the masked-off area of the render-target surface.
Although multiple texture blending can be used to solve this problem, doing so limits the number of other special effects that your application can produce. Using the stencil buffer to apply decals frees up texture blending stages for other effects.

Compositing

Your application can use the stencil buffer to composite 2-D or 3-D images onto a 3-D scene. A mask in the stencil buffer is used to occlude an area of the rendering target surface. Stored 2-D information, such as text or bitmaps, can then be written to the occluded area. Alternately, your application can render additional 3-D primitives to the stencil-masked region of the rendering target surface. It can even render an entire scene.
Games often composite multiple 3-D scenes together. For instance, driving games typically display a rear-view mirror. The mirror contains the view of the 3-D scene behind the driver. It is essentially a second 3-D scene composited with the driver's forward view.

Outlines and Silhouettes

You can use the stencil buffer for more abstract effects, such as outlining and silhouetting.
If your application applies a stencil mask to the image of a primitive that is the same shape but slightly smaller, the resulting image contains only the primitive's outline. The application can then fill the stencil-masked area of the image with a solid color, giving the primitive an embossed look.
If the stencil mask is the same size and shape as the primitive you are rendering, the resulting image contains a hole where the primitive should be. Your application can then fill the hole with black to produce a silhouette of the primitive.

Vertex Tweening

This section contains information about vertex tweening using multiple streams. The following topics are discussed.

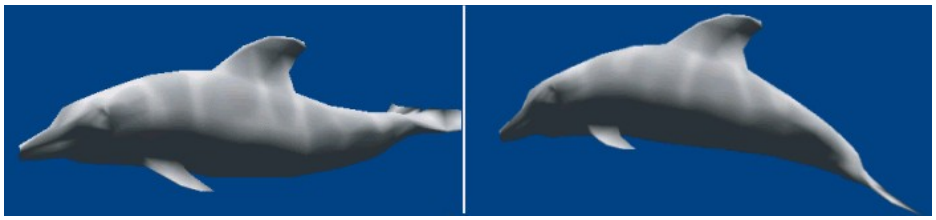
- About Vertex Tweening
- Vertex Tweening Algorithm
- Using Vertex Tweening

About Vertex Tweening

Vertex tweening is used to blend two user-provided positions or normal streams. It can be used only with declarations. Tweening is enabled by setting D3DRS_VERTEXBLEND to D3DVBF_TWEENING. Vertex blending using weights and vertex blending tweening are mutually exclusive.

[C++]
The C++ DolphinVS Sample uses vertex tweening to animate a dolphin so that it appears to move through the water. The images below show the dolphin at two different points. Notice that the dolphin's tail has gone from an up position to a down position and its nose points downward more in the right image.

[Visual Basic]
The Visual Basic Dolphin Sample uses vertex tweening to animate a dolphin so that it appears to move through the water. The images below show the dolphin at two different points. Notice that the dolphin's tail has gone from an up position to a down position and its nose points downward more in the right image.



‹{bmc dolphin1.gif}›

Vertex Tweening Algorithm

Tweening is performed before lighting and clipping. The resulting vertex position (normal) is computed as follows:
POSITION = POSITION1 * (1.0 - TWEENFACTOR) + POSITION2 * TWEENFACTOR, and likewise for normals.

Using Vertex Tweening

The following topics are discussed.

- Determining Support for Vertex Tweening
- Setting Vertex Declaration
- Fixed-Function Tweening
- Vertex Shader Tweening

Determining Support for Vertex Tweening

[C++]
To determine if Microsoft® Direct3D® supports vertex tweening, check for the D3DVTXPCAPS_TWEENING flag in the **VertexProcessingCaps** member of the

D3DCAPS8 structure. The following code example uses the **IDirect3DDevice8::GetDeviceCaps** method to determine if tweening is supported.

```
//
// This example assumes that m_d3dDevice is
// a valid pointer to a IDirect3DDevice8 interface.
//

D3DCAPS8 d3dCaps;

m_d3dDevice->GetDeviceCaps( &d3dCaps );
if( 0 != d3dCaps.VertexProcessingCaps & D3DVTXPCAPS_TWEENING )
   //Vertex tweening is supported.
```

[Visual Basic]

To determine if Microsoft® Direct3D® supports vertex tweening, check for the D3DVTXPCAPS_TWEENING flag in the **VertexProcessingCaps** member of the **D3DCAPS8** structure. The following code example uses the **Direct3DDevice8.GetDeviceCaps** method to determine if tweening is supported.

```
'
' This example assumes that d3dDevice
' is a valid Direct3DDevice object.
'
Dim d3dCaps As D3DCAPS8

d3dDevice.GetDeviceCaps d3dCaps
If 0 <> (d3dCaps.VertexProcessingCaps & D3DVTXPCAPS_TWEENING) Then
   //Vertex tweening is supported.
End If
```

Setting Vertex Declaration

[C++]

To use vector tweening, you must first set up a custom vertex type that uses a second normal or a second position. The following code example shows a sample declaration that includes both a second point and a second position.

```
struct TEX_VERTEX
{
   D3DVECTOR position;
   D3DVECTOR normal;
   D3DVECTOR position2;
   D3DVECTOR normal2;
};
```

```
//Create a vertex buffer with the type TEX_VERTEX.
```

The next step is to set the current declaration. The code example below shows how to do this.

```
DWORD decl[]
{
    D3DVSD_STREAM(0),
    D3DVSD_REG( D3DVSDE_POSITION, D3DVSDT_FLOAT3 ) // Position 1
    D3DVSD_REG( D3DVSDE_NORMAL, D3DVSDT_FLOAT3 )   // Normal 1
    D3DVSD_REG( D3DVSDE_POSITION2, D3DVSDT_FLOAT3) // Position 2
    D3DVSD_REG( D3DVSDE_NORMAL2, D3DVSDT_FLOAT3 )  // Normal 2
    D3DVSD_END()
};
```

[Visual Basic]

To use vector tweening, you must first set up a custom vertex type that uses a second normal or a second position. The following code example shows a sample declaration that includes both a second point and a second position.

```
Private Type TEX_VERTEX
    Position As D3DVECTOR
    Normal As D3DVECTOR
    Position2 As D3DVECTOR
    Normal2 As D3DVECTOR
};

' Create a vertex buffer with the type TEX_VERTEX.
```

The next step is to set the current declaration. The code example below shows how to do this.

```
Dim decl(5) As Long

decl(0) = D3DVSD_STREAM(0)
decl(1) = D3DVSD_REG( D3DVSDE_POSITION, D3DVSDT_FLOAT3 ) 'Position 1
decl(2) = D3DVSD_REG( D3DVSDE_NORMAL, D3DVSDT_FLOAT3 )   'Normal 1
decl(3) = D3DVSD_REG( D3DVSDE_POSITION2, D3DVSDT_FLOAT3) 'Position 2
decl(4) = D3DVSD_REG( D3DVSDE_NORMAL2, D3DVSDT_FLOAT3 )  'Normal 2
decl(5) = D3DVSD_END()
```

For more information on creating a custom vertex type and a vertex buffer, see Creating a Vertex Buffer.

For more information on creating a vertex shader declaration, see Vertex Shader Declaration.

Notes

When vertex tweening is enabled, a second position or a second normal must be present in the current declaration.

Fixed-Function Tweening

[C++]

The code example below shows how to implement fixed-function vertex tweening after a vertex type and declaration are set.

```
//Variables used for this example
DWORD handle;
float TweenFactor = 0.3f;
```

The first step is to use the **IDirect3DDevice8::CreateVertexShader** method to create a vertex shader, as shown in the code example below.

```
m_d3dDevice->CreateVertexShader( decl, NULL, &handle, 0 );
```

The first parameter accepted by **CreateVertexShader** is a pointer to a vector shader declaration. This example uses the declaration declared in the topic Setting Vertex Declaration. The second parameter accepts a pointer to a vertex shader function array. This example does not use a vertex shader function, so this parameter is set to NULL. The third parameter accepts a pointer to a vertex shader handle representing the returned vertex. The fourth parameter specifies the usage controls for the vertex shader. You can use the D3DUSAGE_SOFTWAREPROCESSING flag to indicate that the vertex shader should use software vertex processing. This example sets this parameter to zero to indicate that vertex processing should be done in hardware.
The next step is to set the current vertex shader by calling the **IDirect3DDevice8::SetVertexShader** method as shown in the code example below.

```
m_d3dDevice->SetVertexShader( handle );
```

SetVertexShader accepts a handle to a vertex shader. The value for this parameter can be a handle returned by **CreateVertexShader** or an FVF code. This example uses the handle returned from **CreateVertexShader** called in the last step.
The next step is to set the current render state to enable vertex tweening, set the tween factor, and set the stream source. The code example below uses **IDirect3DDevice8::SetRenderState** and **IDirect3DDevice8::SetStreamSource** to accomplish this.

```
m_d3dDevice->SetRenderState( D3DRS_VERTEXBLEND, D3DVBF_TWEENING );
m_d3dDevice->SetRenderState( D3DRS_TWEENFACTOR, *(DWORD*) &TweenFactor );
m_d3dDevice->SetStreamSource( 0 ,vb, 12*sizeof(float));
```

If you have more than one stream specified for the current vertex shader, you need to call **SetStreamSource** for each stream that will be used for the tweening effect. At this point, vertex tweening is enabled. A call to any rendering function automatically has vertex tweening applied to it.

[Visual Basic]
The code example below shows how to implement fixed-function vertex tweening after a vertex type and declaration are set.

```
' Variables used for this example
Dim handle As Long
Dim TweenFactor As Float

TweenFactor = 0.3
```

The first step is to use the **Direct3DDevice8.CreateVertexShader** method to create a vertex shader, as shown in the code example below.

```
Call d3dDevice.CreateVertexShader(decl, ByVal 0, handle, 0)
```

The first parameter accepted by **CreateVertexShader** is a vector shader declaration. This example uses the declaration declared in the topic Setting Vertex Declaration. The second parameter accepts a vertex shader function array. This example does not use a vertex shader function, so this parameter is set to ByVal 0. The third parameter accepts a vertex shader handle representing the returned vertex. The fourth parameter specifies the usage controls for the vertex shader. You can use the D3DUSAGE_SOFTWAREPROCESSING flag to indicate that the vertex shader should use software vertex processing. This example sets this parameter to zero to indicate that vertex processing should be done in hardware.
The next step is to set the current vertex shader by calling the **Direct3DDevice8.SetVertexShader** method as shown in the code example below.

```
Call d3dDevice.SetVertexShader(handle)
```

SetVertexShader accepts a handle to a vertex shader. The value for this parameter can be a handle returned by **CreateVertexShader** or an FVF code. This example uses the handle returned from **CreateVertexShader** called in the last step.
The next step is to set the current render state to enable vertex tweening, set the tween factor, and set the stream source. The code example below uses **Direct3DDevice8.SetRenderState** and **Direct3DDevice8.SetStreamSource** to accomplish this.

```
Call d3dDevice.SetRenderState(D3DRS_VERTEXBLEND, D3DVBF_TWEENING)
Call d3dDevice.SetRenderState(D3DRS_TWEENFACTOR, TweenFactor)
Call d3dDevice.SetStreamSource(0 ,vb, 12*Len(vb(0))
```

If you have more than one stream specified for the current vertex shader, you need to call **SetStreamSource** for each stream that will be used for the tweening effect. At this point, vertex tweening is enabled. A call to any rendering function automatically has vertex tweening applied to it.

Vertex Shader Tweening

[C++]
The code for using vertex shader tweening is similar to the code for Fixed-Function Tweening. The only difference is the call to **IDirect3DDevice8::CreateVertexShader** as shown below.

```
m_d3dDevice->CreateVertexShader( decl, &vsFunction, &handle, 0 );
```

The second parameter for **CreateVertexShader** takes a pointer to a vertex shader function token array. This function can be created by calling any of the following functions.

- **D3DXAssembleShader**
- **D3DXAssembleShaderFromFileA**
- **D3DXAssembleShaderFromFileW**

The code example below assembles a vertex shader from the file AppTween.vsh.

```
D3DXAssembleShaderFromFile( "AppTween.vsh", 0, NULL, &vsFunction, NULL );
```

[Visual Basic]
The code for using vertex shader tweening is similar to the code for Fixed-Function Tweening. The only difference is the call to **Direct3DDevice8.CreateVertexShader** as shown below.

```
Call d3dDevice.CreateVertexShader(decl, vsFunction, handle, 0)
```

The second parameter for **CreateVertexShader** takes a pointer to a vertex shader function token array. This function can be created by calling any of the following functions.

- **D3DX8.AssembleShader**
- **D3DX8.AssembleShaderFromFile**

The code example below assembles a vertex shader from the file AppTween.vsh.

```
Set vsFunction = d3dx.AssembleShaderFromFile( "AppTween.vsh", 0, ErrorLog, vsFunction)
```

For more information on creating a vertex shader function, see Vertex Shader Function.

Colored Lights

[C++]
The color-related members of the **D3DLIGHT8** structure (**Diffuse**, **Specular**, and **Ambient**) are **D3DCOLORVALUE** structures. The colors defined by these structures are RGBA values that generally range from 0.0 to 1.0, with 0.0 being black. Although you will usually want the light color to fall within this range, you can use values outside the range for special effects. For example, you can create a strong light

that washes out a scene by setting the color to large values. You can also set the color to negative values to create a dark light, which actually removes light from a scene. Dark lights are useful for forcing dramatic shadows in scenes and other special effects.

[Visual Basic]
The color-related members of the **D3DLIGHT8** type (**diffuse**, **specular**, **ambient**) are **D3DCOLORVALUE** types. The colors defined by these are RGBA values that generally range from zero to one, with zero being black. Although you will usually want the light color to fall within this range, you can use values outside the range for special effects. For example, you can create a strong light that washes out a scene by setting the color to large values. You can also set the color to negative values to create a dark light, which actually removes light from a scene. Dark lights are useful for forcing dramatic shadows in scenes and other special effects.

Advanced Topics in DirectX Graphics

This section is a guide to using the advanced features of the Microsoft® Direct3D® and Direct3DX APIs. It goes beyond the information covered in Using DirectX Graphics.

Information is presented in the following topics.

- Effects and Techniques
- Higher-Order Primitives
- Matrix Stack
- Pixel Shaders
- Point Sprites
- Vertex Shaders
- X Files

Effects and Techniques

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
This section introduces effects and techniques and discusses how they can be used in your applications. Information is divided into the following sections.

- Introduction to Effects and Techniques
- Effect File Components
- Using Effects and Techniques

Introduction to Effects and Techniques

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Using an effect in Microsoft® Direct3D® allows an application to define many different techniques for a single effect. Each effect contains one or more techniques that controls the rendering of the effect. Each technique can be validated to see if the current device can support it. If the current device does not support a technique, then a different technique can be used.

You can use different techniques to control the complexity of a rendered effect. For example, say you want a pond of water that has rippled waves that reflect light. One technique might render the entire scene with multiple textures and lighting in one pass, while other techniques might use multiple-pass rendering. This can be done so that video cards that don't support single-pass rendering can still render an effect.

You can also adjust effects based on the level of detail needed. An effect that appears in the distance might use a simpler technique to decrease the computational overhead that is incurred when rendering a complex technique.

Another advantage to using effects is that you can modify the effect file without recompiling the source application. This enables you to release a product that makes optimum use of modern video cards. It also enables you to update the effect file to take advantage of video cards of the future.

Effect File Components

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
An effect file defines the techniques that will be used. The basic layout of an effect file starts with one or more declarations and then defines each technique for that effect. The sample below shows a basic effect file that contains two textures and two techniques.

The effect file starts by declaring two textures that will be used for the effect.

```
texture tex0; //First texture
texture tex1; //Second texture
```

The first technique uses one pass to render the scene.

```
technique t0
{
    pass p0
    {
        Texture[0] = <tex0>;
        Texture[1] = <tex1>;

        ColorOp[0] = SelectArg1;
        ColorArg1[0] = Texture;

        ColorOp[1] = Add;
        ColorArg1[0] = Texture;
        ColorArg2[0] = Current;

        ColorOp[2] = Disable;
    }
}
```

The second technique uses multiple passes to render the scene. This technique would be used if the current device did not support single-pass rendering for multiple textures.

```
technique t1
{
    pass p0
    {
        Texture[0] = <tex0>;

        ColorOp[0] = SelectArg1;
        ColorArg1[0] = Texture;
        ColorOp[1] = Disable;
    }

    pass p1
    {
        AlphaBlendEnable = True;
```

```
        SrcBlend = One;
        DestBlend = One;

        Texture[0] = <tex1>;

        ColorOp[0] = SelectArg1;
        ColorArg1[0] = Texture;
        ColorOp[1] = Disable;
    }
}
```

This effect file enables a device that doesn't support single-pass rendering for two textures to use multiple passes to render the textures. For more information on the layout of an effect file, see Effect File Format.

Using Effects and Techniques

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
This section shows how to create and use effects and techniques in an application.

- Creating an Effect
- Validating a Technique
- Rendering with a Technique
- Retrieving Effect and Technique Descriptions
- Sample Effect File

Creating an Effect

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The first step to creating an effect is to compile the effect. The following functions can be used to compile an effect.

- **D3DXCompileEffect**

- **D3DXCompileEffectFromFileA**
- **D3DXCompileEffectFromFileW**

The code example below uses the **D3DXCompileEffectFromFileA** function to load and compile the effect described in the file Water.sha.

```
LPD3DXBUFFER pCompiledEffect;
D3DXCompileEffectFromFile( "Water.sha", &pCompiledEffect, NULL );
```

The first parameter that **D3DXCompileEffectFromFileA** accepts is a pointer to an ANSI string that specifies the file from which to create the effect. The second parameter accepts the address of a pointer to an **ID3DXBuffer** interface. This buffer will contain the compiled effect. The final parameter that **D3DXCompileEffectFromFileA** accepts is the address of a pointer to an **ID3DXBuffer** interface. This buffer will contain returned ASCII error message. This example ignores the errors and sets this parameter to NULL.

After compiling an effect, use the **D3DXCreateEffect** function to obtain a pointer to an **ID3DXEffect** interface as shown in the code sample below.

```
ID3DXEffect * pEffect;
D3DXCreateEffect( g_d3dDevice, pCompiledEffect->GetBufferPointer(),
            pCompiledEffect->GetBufferSize(), 0, &pEffect);
pCompiledEffect->Release();
```

If this function succeeds, then pEffect contains a valid pointer to **ID3DXEffect** interface. When finished with the **ID3DXBuffer** that stored the compiled effect, be sure to call the **IUnknown::Release** method to avoid a memory leak.

Validating a Technique

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Before using a technique, it is good to verify that the device can support it. To validate a technique, use the **ID3DXTechnique::Validate** method as show in the code sample below.

```
LPD3DXTECHNIQUE pTechnique;
pEffect->GetTechnique(0, &pTechnique);
if ( pTechnique->Validate() != D3D_OK)
   //Technique is not valid on current device.
```

Note

It is strongly recommended that you validate a technique before using it. Using invalid techniques has unknown effects.

Rendering with a Technique

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The following sample shows how an application would render a technique. After finding a valid technique the application can call the **ID3DXTechnique::Begin** method to start the application of the technique as shown in the code example below.

```
UINT uPasses;
pTechnique->Begin(&uPasses);
```

The only parameter that **Begin** accepts is the address of a unsigned integer. After **Begin** is called, this integer will contain the number of passes used for the technique.

The next step is to render each pass individually. To render a pass, call the **ID3DXTechnique::Pass** method. The code example below uses a loop to render all the passes required for a technique.

```
for(UINT uPass = 0; uPass < uPasses; uPass++)
{
    pTechnique->Pass(uPass);
    g_d3dDevice->SetStreamSource( 0, pvbVertices, sizeof(WATER_VERTEX) );
    g_d3dDevice->SetIndices( pIB, 0 );
    g_d3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLESTRIP, 0,
                                       uVertices, 0, uIndices -2 );
}
```

The call to **Pass** sets up the necessary states to render the scene.

Note

The call to **ID3DXTechnique::Begin** must be inside a **IDirect3DDevice8::BeginScene** / **IDirect3DDevice8::EndScene** block.

Retrieving Effect and Technique Descriptions

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
To retrieve information about an effect, use the **ID3DXEffect::GetDesc** method and pass a **D3DXEFFECT_DESC** structure for the parameter. The **D3DXEFFECT_DESC** structure contains members that describe an effect.

To retrieve information about a technique, use the **ID3DXTechnique::GetDesc** method and pass a **D3DXTECHNIQUE_DESC** structure for the parameter. The **D3DXTECHNIQUE_DESC** structure contains members that describe the technique.

Sample Effect File

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The following effect file is used for the Water Sample.

The file starts by declaring two matrices, three textures, and a vertex shader that will be used for this effect.

```
matrix mID;     // Identity transform
matrix mENV;    // Environment map transform

texture tFLR;   // Floor texture
texture tCAU;   // Caustic texture
texture tENV;   // Cubic environment map

vertexshader vs =  decl { fvf xyz | normal | diffuse | tex1; };
```

The first technique renders the water scene using both the floor texture and caustic texture. Both the water and the floor reflect light. This technique uses a single pass and four texture stage states.

```
technique T0
{
   vertexshader = <vs>;

   pass P0
   {
      Lighting      = True;
      SpecularEnable = True;

      // Stage0
      ColorOp[0]   = SelectArg1;
      ColorArg1[0] = Texture;
```

```
AlphaOp[0]   = SelectArg2;
AlphaArg2[0] = Current;

MinFilter[0] = Linear;
MagFilter[0] = Linear;
MipFilter[0] = Point;

Texture[0] = <tFLR>;
TextureTransform[0] = <mID>;
TextureTransformFlags[0] = Count2;


// Stage1
ColorOp[1]   = Add;
ColorArg1[1] = Texture;
ColorArg2[1] = Current;
AlphaOp[1]   = SelectArg2;
AlphaArg2[1] = Current;

MinFilter[1] = Linear;
MagFilter[1] = Linear;
MipFilter[1] = Point;

Texture[1] = <tCAU>;
TextureTransform[1] = <mID>;
TextureTransformFlags[1] = Count2;


// Stage2
ColorOp[2]   = BlendCurrentAlpha;
ColorArg1[2] = Texture;
ColorArg2[2] = Current;
AlphaOp[2]   = Disable;

MinFilter[2] = Linear;
MagFilter[2] = Linear;
MipFilter[2] = Point;

Texture[2] = <tENV>;
TextureTransform[2] = <mENV>;
TextureTransformFlags[2] = Count3;
TexCoordIndex[2] = CameraSpaceReflectionVector;


// Stage3
ColorOp[3] = Disable;
```

```
        AlphaOp[3] = Disable;
    }
}
```

The second technique renders the water scene using both the floor texture and caustic texture. Both the floor and water reflect light. This technique uses two passes to render the scene.

```
technique T1
{
    vertexshader = <vs>;

    pass P0
    {
        // Stage0
        ColorOp[0]   = SelectArg1;
        ColorArg1[0] = Texture;
        AlphaOp[0]   = Disable;

        MinFilter[0] = Linear;
        MagFilter[0] = Linear;
        MipFilter[0] = Point;

        Texture[0] = <tFLR>;
        TextureTransform[0] = <mID>;
        TextureTransformFlags[0] = Count2;


        // Stage1
        ColorOp[1]   = Add;
        ColorArg1[1] = Texture;
        ColorArg2[1] = Current;
        AlphaOp[1]   = Disable;

        MinFilter[1] = Linear;
        MagFilter[1] = Linear;
        MipFilter[1] = Point;

        Texture[1] = <tCAU>;
        TextureTransform[1] = <mID>;
        TextureTransformFlags[1] = Count2;


        // Stage2
        ColorOp[2] = Disable;
        AlphaOp[2] = Disable;
```

```
    }


    pass P1
    {
        SrcBlend  = One;
        DestBlend = InvSrcAlpha;

        Lighting        = True;
        SpecularEnable  = True;
        AlphaBlendEnable = True;


        // Stage0
        ColorOp[0]   = BlendDiffuseAlpha;
        ColorArg1[0] = Texture;
        ColorArg2[0] = Diffuse;
        AlphaOp[0]   = SelectArg2;
        AlphaArg2[0] = Diffuse;

        MinFilter[0] = Linear;
        MagFilter[0] = Linear;
        MipFilter[0] = Point;

        Texture[0] = <tENV>;
        TextureTransform[0] = <mENV>;
        TextureTransformFlags[0] = Count3;
        TexCoordIndex[0] = CameraSpaceReflectionVector;


        // Stage1
        ColorOp[1] = Disable;
        AlphaOp[1] = Disable;
    }
}
```

The third technique renders the water scene using both the floor texture and caustic texture. The water surface reflects light. This effect uses three passes to render the scene.

```
technique T2
{
    vertexshader = <vs>;

    pass P0
    {
```

```
    // Stage0
    ColorOp[0]   = SelectArg1;
    ColorArg1[0] = Texture;
    AlphaOp[0]   = Disable;

    MinFilter[0] = Linear;
    MagFilter[0] = Linear;
    MipFilter[0] = Point;

    Texture[0] = <tFLR>;
    TextureTransform[0] = <mID>;
    TextureTransformFlags[0] = Count2;

    // Stage1
    ColorOp[1] = Disable;
    AlphaOp[1] = Disable;
}


pass P1
{
    SrcBlend  = One;
    DestBlend = One;

    Lighting        = False;
    SpecularEnable  = True;
    AlphaBlendEnable = True;


    // Stage0
    ColorOp[0]   = Add;
    ColorArg1[0] = Texture;
    ColorArg2[0] = Current;
    AlphaOp[0]   = Disable;

    MinFilter[0] = Linear;
    MagFilter[0] = Linear;
    MipFilter[0] = Point;

    Texture[0] = <tCAU>;
    TextureTransform[0] = <mID>;
    TextureTransformFlags[0] = Count2;


    // Stage1
    ColorOp[1] = Disable;
```

```
    AlphaOp[1] = Disable;
  }


  pass P2
  {
    SrcBlend  = One;
    DestBlend = InvSrcAlpha;

    Lighting        = True;
    SpecularEnable  = True;
    AlphaBlendEnable = True;


    // Stage0
    ColorOp[0]   = BlendDiffuseAlpha;
    ColorArg1[0] = Texture;
    ColorArg2[0] = Diffuse;
    AlphaOp[0]   = SelectArg2;
    AlphaArg2[0] = Diffuse;

    MinFilter[0] = Linear;
    MagFilter[0] = Linear;
    MipFilter[0] = Point;

    Texture[0] = <tENV>;
    TextureTransform[0] = <mENV>;
    TextureTransformFlags[0] = Count3;
    TexCoordIndex[0] = CameraSpaceReflectionVector;


    // Stage1
    ColorOp[1] = Disable;
    AlphaOp[1] = Disable;
  }
}
```

The fourth technique renders the water scene using both the floor texture and caustic texture. This effect uses a single pass to render the scene.

```
technique T3
{
  vertexshader = <vs>;

  pass P0
```

```
    {
        Lighting       = True;
        SpecularEnable = True;


        // Stage0
        ColorOp[0]   = SelectArg1;
        ColorArg1[0] = Texture;
        AlphaOp[0]   = Disable;

        MinFilter[0] = Linear;
        MagFilter[0] = Linear;
        MipFilter[0] = Point;

        Texture[0]   = <tFLR>;


        // Stage1
        ColorOp[1]   = Add;
        ColorArg1[1] = Texture;
        ColorArg2[1] = Current;
        AlphaOp[1]   = Disable;

        MinFilter[1] = Linear;
        MagFilter[1] = Linear;
        MipFilter[1] = Point;

        Texture[1]   = <tCAU>;


        // Stage2
        ColorOp[2]   = Disable;
        AlphaOp[2]   = Disable;
    }
}
```

The fifth technique renders the water scene using both the floor and caustic textures. There is no reflection in this technique. This effect uses two passes to render the scene.

```
technique T4
{
    vertexshader = <vs>;

    pass P0
```

```
{
    // Stage 0
    ColorOp[0]   = SelectArg1;
    ColorArg1[0] = Texture;
    ColorArg2[0] = Diffuse;

    AlphaOp[0]   = SelectArg1;
    AlphaArg1[0] = Texture;
    AlphaArg2[0] = Diffuse;

    MinFilter[0] = Linear;
    MagFilter[0] = Linear;
    MipFilter[0] = Point;

    Texture[0]   = <tFLR>;


    // Stage 1
    ColorOp[1]   = Disable;
    AlphaOp[1]   = Disable;
}


pass P1
{
    SrcBlend  = One;
    DestBlend = One;

    Lighting        = True;
    SpecularEnable   = True;
    AlphaBlendEnable = True;


    // Stage 0
    ColorOp[0]   = SelectArg1;
    ColorArg1[0] = Texture;
    ColorArg2[0] = Current;

    AlphaOp[0]   = SelectArg1;
    AlphaArg1[0] = Texture;
    AlphaArg2[0] = Current;

    MinFilter[0] = Linear;
    MagFilter[0] = Linear;
    MipFilter[0] = Point;
```

```
      Texture[0]   = <tCAU>;


      // Stage 1
      ColorOp[1]   = Disable;
      AlphaOp[1]   = Disable;
   }
}
```

The sixth and final technique renders the water scene with just the floor texture. There is no reflection in this technique. This effect uses a single pass to render the scene.

```
technique T5
{
   vertexshader = <vs>;

   pass P0
   {
      Lighting      = True;
      SpecularEnable = True;


      // Stage 0
      ColorOp[0]   = SelectArg1;
      ColorArg1[0] = Texture;
      ColorArg2[0] = Current;

      AlphaOp[0]   = SelectArg1;
      AlphaArg1[0] = Texture;
      AlphaArg2[0] = Current;

      MinFilter[0] = Linear;
      MagFilter[0] = Linear;
      MipFilter[0] = None;

      Texture[0]   = <tFLR>;

      // Stage 1
      ColorOp[1]   = Disable;
      AlphaOp[1]   = Disable;
   }
}
```

Higher-Order Primitives

This section introduces higher-order primitives and discusses how they can be used in your applications. Information is divided into the following sections.

- What are Higher-Order Primitives?
- Improved Quality through Resolution Enhancement
- Direct Mapping from Spline-Based Tools
- Using Higher-Order Primitives

What are Higher-Order Primitives?

Microsoft® Direct3D® for Microsoft DirectX® 8.0 supports points, lines, triangles, and grid primitives. These have been extended to support higher-order interpolation beyond linear. While triangles and lines have spatial extent, until now they were both rendered using linear interpolation. In DirectX 8.0, Direct3D supports rendering of these primitive types using higher order, up to quintic, interpolation. Furthermore, a new quad primitive type is now supported. This new type can also be rendered with higher-order interpolation. This feature is primarily driven by requirements for animation and rendering of characters. It can also be used for other surfaces such as terrain or water.

Higher-order primitives support higher-order interpolation when transmitted to the application programming interface (API) as lists, strips, fans, or indexed meshes. This is achieved by using additional information encoded in the vertices themselves. For example, normal vectors can be used to define tangent planes at the vertices to enable cubic interpolation. Most implementations support higher-order interpolation by tessellation into planar triangles. The tessellation step is applied logically before the vertex shader stage. Because the vertex shader API does not impose semantics on its input data, a special mechanism is provided to identify the vertex stream component that represents the position, and optionally the normal vector. All other components are interpolated accordingly.

Improved Quality through Resolution Enhancement

Current primitives are not ideal for representing smooth surfaces. Higher-order interpolation methods, such as cubic polynomials, allow more accurate calculations in rendering curved shapes. This provides increased realism by reducing or eliminating faceting artifacts visible on silhouette edges or on specular surface lighting. Furthermore, when tessellation occurs on the chip, the tessellated triangles do not impact the bus bandwidth. In many cases, a small amount of tessellation can provide improvements in image quality with minimal performance impact.

Microsoft® Direct3D® for Microsoft DirectX® 8.0 provides a simple way to apply resolution enhancement to content created by existing polygon-oriented tools and art

pipelines. The application need only provide a desired level of tessellation, and transmit the data using standard triangle syntax that includes normal vectors.

Direct Mapping from Spline-Based Tools

Many current authoring tools support higher-order primitives to enable more powerful modeling operations than are commonly provided for with planar triangle meshes. When used efficiently, so that the number of patches generated is reasonable, such tools can produce content that can be rendered directly by the application programming interface (API). To meet this requirement, a new entry point has been added that interprets the incoming vertex data stream as a 2-D array of control points and tessellates it to the desired resolution.

Using Higher-Order Primitives

This section shows you how to use higher-order primitives in your application.

- Determining Higher-Order Primitive Support
- Drawing Patches

Determining Higher-Order Primitive Support

[C++]
The **DevCaps** member of **D3DCAPS8** can be queried to determine the level of support for operations involving higher-order primitives. The following table lists the device capabilities related to higher-order primitives in Microsoft® DirectX® 8.0.

Device capability	Description
D3DDEVCAPS_NPATCHES	Device supports N-patches.
D3DDEVCAPS_QUINTICRTPATCHES	Device supports quintic béziers and B-splines.
D3DDEVCAPS_RTPATCHES	Device supports rectangular and triangular (RT) patches.
D3DDEVCAPS_RTPATCHHANDLEZERO	RT-patches might be drawn efficiently using handle zero.

[Visual Basic]
The **DevCaps** member of **D3DCAPS8** contains values from the **CONST_D3DDEVCAPSFLAGS** enumeration which can be queried to determine the level of support for operations involving higher-order primitives. The following table lists the device capabilities related to higher-order primitives in Microsoft® DirectX® 8.0.

Device capability	Description

D3DDEVCAPS_NPATCHES	Device supports N-patches.
D3DDEVCAPS_QUINTICRTPATCHES	Device supports quintic béziers and B-splines.
D3DDEVCAPS_RTPATCHES	Device supports rectangular and triangular (RT) patches.
D3DDEVCAPS_RTPATCHHANDLEZERO	RT-patches might be drawn efficiently using handle zero.

Note that D3DDEVCAPS_RTPATCHHANDLEZERO does not mean that a patch with handle zero can be drawn. A handle zero patch can always be drawn, whether this device capability is set or not. When this capability is set, the hardware architecture does not require caching of any information and that uncached patches (handle zero) be drawn as efficiently as cached ones.

Drawing Patches

[C++]
Microsoft® DirectX® 8.0 supports two types of higher order primitives, or patches. These are referred to as N-Patches and Rect/Tri patches. N-Patches can be rendered using any triangle rendering call by enabling D3DRS_PATCHSEGMENTS to a value greater than 1.0. Rect/Tri patches must be rendered using the following explicit entry points.

You can use the following methods to draw patches in Microsoft Direct3D® for DirectX.

- **IDirect3DDevice8::DrawRectPatch**
- **IDirect3DDevice8::DrawTriPatch**

DrawRectPatch draws a rectangular high-order patch specified by the *pRectPatchInfo* parameter using the currently set streams. The *Handle* parameter is used to associate the patch with a handle, so that the next time the patch is drawn, there is no need to respecify *pRectPatchInfo*. This makes it possible to precompute and cache forward difference coefficients or other information, which in turn enables subsequent calls to **DrawRectPatch** using the same handle to execute efficiently.

It is intended that for static patches, an application would set the vertex shader and appropriate streams, supply patch information in the *pRectPatchInfo* parameter, and specify a handle so that Direct3D can capture and cache information. The application can then call **DrawRectPatch** subsequently with *pRectPatchInfo* set to NULL to efficiently draw the patch. When drawing a cached patch, the currently set streams are ignored. However, it is possible to override the cached *pNumSegs* by specifying new values for *pNumSegs*. Also, it is required to set the same vertex shader when rendering a cached patch as was set when it was captured.

For dynamic patches, the patch data changes for every rendering of the patch, so it is not efficient to cache information. The application can convey this to Direct3D by setting *Handle* to 0. In this case, Direct3D draws the patch using the currently set streams and the *pNumSegs* values and does not cache any information. It is not valid to simultaneously set *Handle* to 0 and *pPatch* to NULL.

By respecifying *pRectPatchInfo* for the same handle, the application can overwrite the previously cached information.

If *pNumSegs* is set to NULL, then the tessellator uses D3DRS_PATCHSEGMENTS to control the amount of tessellation. In this case, obviously, the same tessellation is used for all sides. There are no special methods for N-patches. Tessellation is simply controlled by D3DRS_PATCHSEGMENTS. It is necessary to supply normals.

DrawTriPatch is similar to **DrawRectPatch** except that it draws a triangular high-order patch.

[Visual Basic]
You can use the following methods to draw patches in Microsoft® Direct3D® for Microsoft DirectX® 8.0.

- **Direct3DDevice8.DrawRectPatch**
- **Direct3DDevice8.DrawTriPatch**

DrawRectPatch draws a rectangular high-order patch specified by the *Surface* parameter using the currently set streams. The *Handle* parameter is used to associate the patch with a handle, so that the next time the patch is drawn, there is no need to respecify *Surface*. This makes it possible to precompute and cache forward difference coefficients or other information, which in turn enables subsequent calls to **DrawRectPatch** using the same handle to execute efficiently.

It is intended that for static patches, an application would set the vertex shader and appropriate streams, supply patch information in the *Surface* parameter, and specify a handle so that Direct3D can capture and cache information. The application can then call **DrawRectPatch** with *Surface* set to ByVal 0 to efficiently draw the patch. When drawing a cached patch, the currently set streams are ignored. However, you can override the cached *NumSegments* by specifying new values for *NumSegments*. Also, it is required to set the same vertex shader when rendering a cached patch as was set when it was captured.

For dynamic patches, the patch data changes for every rendering of the patch, so it is not efficient to cache information. The application can convey this to Direct3D by setting *Handle* to 0. In this case, Direct3D draws the patch using the currently set streams and the *NumSegments* values and does not cache any information. It is not valid to simultaneously set *Handle* to 0 and *Surface* to ByVal 0.

By respecifying *Surface* for the same handle, the application can overwrite the previously cached information.

If *NumSegments* is set to ByVal 0, then the tessellator uses D3DRS_PATCHSEGMENTS to control the amount of tessellation. In this case, obviously, the same tessellation is used for all sides. There are no special methods for N-patches. Tessellation is simply controlled by D3DRS_PATCHSEGMENTS. It is necessary to supply normals.

DrawTriPatch is similar to **DrawRectPatch**, except that it draws a triangular high-order patch.

Matrix Stack

[Visual Basic]
The information in this section pertains only to applications written in C and C++.

[C++]
The Direct3DX utility library provides a matrix stack through a COM object.

This section provides information on the following topics.

- What Is a Matrix Stack?
- Implementing a Scene Hierarchy

What Is a Matrix Stack?

[Visual Basic]
The information in this section pertains only to applications written in C and C++.

[C++]
The Direct3DX utility library provides the **ID3DXMatrixStack** interface. It supplies a mechanism to enable matrices to be pushed onto and popped off of a matrix stack. Implementing a matrix stack is an efficient way to track matrices while traversing a transform hierarchy.

The Direct3DX utility library uses a matrix stack to store transformations as matrices. The various methods of the **ID3DXMatrixStack** interface deal with the current matrix, or the matrix located on top of the stack. You can clear the current matrix with the **ID3DXMatrixStack::LoadIdentity** method. To explicitly specify a certain matrix to load as the current transformation matrix, use the **ID3DXMatrixStack::LoadMatrix** method. Then you can call either the **ID3DXMatrixStack::MultMatrix** method or the

ID3DXMatrixStack::MultMatrixLocal method to multiply the current matrix by the specified matrix.

The **ID3DXMatrixStack::Pop** method enables you to return to the previous transformation matrix and the **ID3DXMatrixStack::Push** method adds a transformation matrix to the stack.

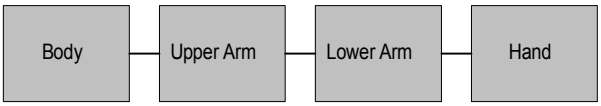
The individual matrices on the matrix stack are represented as 4×4 homogeneous matrices, defined by the Direct3DX utility library **D3DXMATRIX** structure.

Implementing a Scene Hierarchy

[Visual Basic]
The information in this section pertains only to applications written in C and C++.

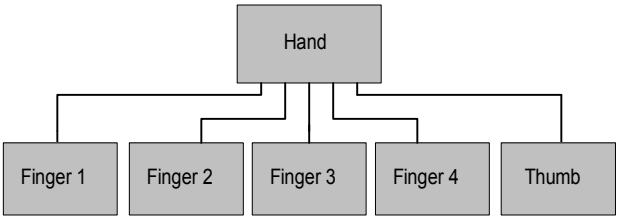
[C++]
A matrix stack simplifies the construction of hierarchical models, in which complicated objects are constructed from a series of simpler objects.

A scene, or transform, hierarchy is usually represented by a tree data structure. Each node in the tree data structure contains a matrix. A particular matrix represents the change in coordinate systems from the node's parent to the node. For example, if you are modeling a human arm, you might implement the following hierarchy.



In this hierarchy, the Body matrix places the body in the world. The UpperArm matrix contains the rotation of the shoulder, the LowerArm matrix contains the rotation of the elbow, and the Hand matrix contains the rotation of the wrist. To determine where the hand is relative to the world, you simply multiply all the matrices from Body down through Hand together.

The previous hierarchy is overly simplistic, because each node has only one child. If you begin to model the hand in more detail, you will probably add fingers and a thumb. Each digit can be added to the hierarchy as children of Hand.



If you traverse the complete graph of the arm in depth-first order—traversing as far down one path as possible before moving on to the next path—to draw the scene, you perform a sequence of segmented rendering. For example, to render the hand and fingers, you implement the following pattern.

1. Push the Hand matrix onto the matrix stack.
2. Draw the hand.
3. Push the Thumb matrix onto the matrix stack.
4. Draw the thumb.
5. Pop the Thumb matrix off the stack.
6. Push the Finger 1 matrix onto the matrix stack.
7. Draw the first finger.
8. Pop the Finger 1 matrix off the stack.
9. Push the Finger 2 matrix onto the matrix stack. You continue in this manner until all the fingers and thumb are rendered.

After you have completed rendering the fingers, you would pop the Hand matrix off the stack.

You can follow this basic process in code with the following examples. When you encounter a node during the depth-first search of the tree data structure, push the matrix onto the top of the matrix stack.

```
MatrixStack->Push();
MatrixStack->MultMatrix(pNode->matrix);
```

When you are finished with a node, pop the matrix off the top of the matrix stack.

```
MatrixStack->Pop();
```

In this way, the matrix on the top of the stack always represents the world-transform of the current node. Therefore, before drawing each node, you should set the Microsoft® Direct3D® matrix.

```
pD3DDevice->SetTransform(D3DTS_WORLDMATRIX(0), *MatrixStack->GetTop());
```

For more information on the specific methods that you can perform on a Direct3DX matrix stack, see the **ID3DXMatrixStack** reference topic.

Pixel Shaders

This section describes pixel shaders, the update to the pixel processing operations supported by Microsoft® DirectX® 8.0, and how they are used in Microsoft Direct3D® applications. The following topics are discussed.

- Introduction to Pixel Shaders
- Pixel Shader Basics
- Pixel Shader Scope
- Pixel Shader Data Flow
- Using Pixel Shaders
- Pixel Shader Examples

All pixel shader code and syntax used in this documentation is based on and compatible with the Direct3DX pixel shader assembler. Before reading this section, ensure that you are familiar with the pixel shader assembler by studying the Pixel Shader Assembler Reference.

Introduction to Pixel Shaders

The following example illustrates a way to introduce pixel shaders and their syntax as understood by the Direct3DX pixel shader assembler.

```
ps.1.0        // DirectX8 version.
tex t0        // Declare the texture.
tex t1        // Declare the light map.

mov r0, t0
mul r0, r0, t1  // Modulate the light map.
mul r0, r0, v0  // Modulate the diffuse.
add r0, r0, v1  // Add the specular.
```

This pixel shader declares a base texture with a light map, diffuse color, and specular add.

Pixel shaders control the color and alpha blending operations and the texture addressing operations. Microsoft® Direct3D® for Microsoft DirectX® 8.0 defines a procedural paradigm for defining pixel and texture blending operations. This model includes a definition of a virtual machine architecture and a set of instructions and modifiers that define its behavior. This paradigm does not necessarily imply a processor-based implementation.

The pixel shaders extend and generalize the multitexture capabilities of DirectX 6.0 and 7.0 in the following ways.

- A set of general read/write registers is added to enable more flexible expression than can be supported by the serial cascade using only D3DTA_CURRENT. This would require the specification of a separate result register argument for each stage.
- The D3DTOP_MODULATE2X and D3DTOP_MODULATE4X texture operations are broken into separate modifiers applicable to any instruction orthogonally. This eliminates the need for separate D3DTOP_MODULATE and D3DTOP_MODULATE2X operations.
- The bias and unbias texture operation modifiers are orthogonal. This eliminates the need for separate add and add bias operations.
- An optional third argument is added to modulate add, so the procedural pixel shader can do arg1 ∗ arg2 + arg0. This eliminates the D3DTOP_MODULATEALPHA_ADDCOLOR and D3DTOP_MODULATECOLOR_ADDALPHA texture operations. In addition, an optional third argument is added to the blend operation, so the procedural pixel shader can use arg0 as the blend proportion between arg1 and arg2. This eliminates the D3DTOP_BLENDDIFFUSEALPHA, D3DTOP_BLENDTEXTUREALPHA, D3DTOP_BLENDFACTORALPHA, D3DTOP_BLENDTEXTUREALPHAPM, and D3DTOP_BLENDCURRENTALPHA texture operations.
- Texture address modifying operations, such as D3DTOP_BUMPENVMAP, are broken out from the color and alpha operations and defined as a third operation type specifically for operating on texture addresses.

To support this increased flexibility more efficiently , the application programming interface (API) syntax is changed from **DWORD** pairs to an ASCII assemble code syntax. This exposes the functionality offered by procedural pixel shaders.

Note

When you use pixel shaders, specular add is no longer specifically controlled by a render state, and it is up to the pixel shader to implement if needed. However, fog blending is still applied by the fixed-function syntax.

For further details and information on how to use the Direct3DX pixel shader assembler, see Pixel Shader Assembler Reference.

Pixel Shader Basics

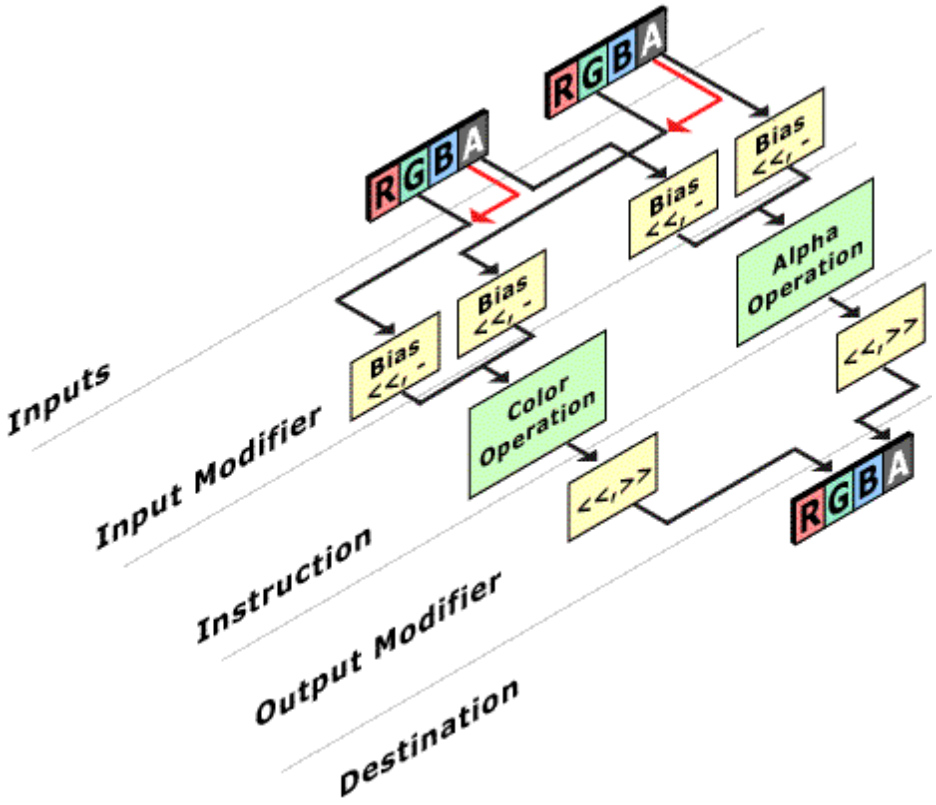
This section provides basic information that you need to know when working with pixel shaders.

- Pixel Shader Arithmetic Logic Unit
- Pixel Shader Instruction Pairing
- Pixel Shader Order of Operations
- Pixel Shader Texture Declarations

- Pixel Shader Texture Addressing
- Pixel Shader Lighting

Pixel Shader Arithmetic Logic Unit

The following diagram illustrates the pixel shader arithmetic logic unit (ALU).



Pixel Shader Instruction Pairing

The processor that executes the pixel shader has two parallel pipelines, one for vector processing (RGB), and one for scalar processing (alpha). Using these pipelines in parallel results in substantially better processor utilization and performance by reducing the total number of clock cycles required. In essence, this affects the pixel fill rate performance.

The output masks can affect how the two pipelines are allocated, but there can be ambiguities in the order of operations unless explicit pairing syntax is used. The following table illustrates the output masks and their associated pipe.

Output mask **Pipe**

.a	The operation is in the scalar pipe.
.rgb	The operation is in the vector pipe.
.rgba	The operation is in both the scalar and vector pipes. It already counts as a pair.

Pairing is indicated by a plus sign (+) preceding the second instruction of the pair.

```
   mul r0.rgb, t0, v0   // Component-wise multiply the colors,
 + add r0.a,   t0, v0   // but add the alpha components.
```

The dot product instructions are a special case. They are vector operations and always executed in the vector pipeline. Therefore, you can specify a different instruction in the scalar pipe and still get pairing, as shown in the following pixel shader example code.

```
dp3 r0.rgb, t0, v0
mul r0.a,   t0, v0
```

You cannot specify a dot product operation as a scalar instruction without a dot product operations as a vector instruction.

Pixel Shader Order of Operations

The order of operations for processing pixel instructions and modifiers is defined as follows: the input or argument modifier is applied, the core instruction is executed, the output or instruction modifier is applied, and then the result is written with respect to the output write mask.

For the input or argument modifiers, the ordering is as follows: bias is applied, scaling is applied, and then the negate or complement step is performed. Alpha replication is also performed at this stage. For the output or instruction modifiers, the scale/shift operations have higher precedence than the saturation step.

The net ordering of operations is defined by the following list.

1. The Argument Modifiers are applied.
2. The core instruction is executed.
3. The Instruction Modifiers are applied.
4. The Output Write Masks are applied.

Pixel Shader Texture Declarations

Texture declarations (texture addressing instructions) are a separate set of pixel shader instructions, as distinguished from the color and alpha blending instructions. These instructions must appear at the top of the pixel shader and can use only previously defined values. Texture addressing instructions can also perform operations. For example, you can use them to modify texture coordinates used as addresses by sampling stages or to enable perturbations effects.

For more information, see Texture Addressing Instructions.

Pixel Shader Texture Addressing

The following diagram illustrates the mechanics of pixel shader texture addressing.

[C++]
Texture registers t0, t1, t2, and t3 result from sampling the four input textures using the texture coordinates in conjunction with the texture operations. The mapping of texture coordinates set to texture registers is controlled by the **D3DTSS_TEXCOORDINDEX** texture stage state. By default, all texture stages receive texture coordinates from coordinate set 0. The positions of the instructions are required to be in order because they can use results only from previous texture registers.

[Visual Basic]
Texture registers t0, t1, t2, and t3 result from sampling the four input textures using the texture coordinates in conjunction with the texture operations. The mapping of texture coordinates set to texture registers is controlled by the **D3DTSS_TEXCOORDINDEX** texture stage state. By default, all texture stages receive texture coordinates from coordinate set 0. The positions of the instructions are required to be in order because they can use results only from previous texture registers.

When you use dependent textures, any texture can be used to perturb the coordinates of a higher-numbered texture; for example, t1 can be used to perturb the coordinates of t2.

The Direct3DX pixel shader assembler provides both texture address operators and modifiers. For details, see Pixel Shader Assembler Reference.

The following pixel shader code shows a texture address shader by programming a bump environment map on a lit base texture.

```
ps.1.0            // DirectX8 Version.
tex t0            // Declare the texture.
tex t1            // Declare the bump map.
texbem t3, t1      // Perturb and then sample
                // the environment map.
mul r0, v0, t0     // Modulate the diffuse.
add r0, r0, t3     // Add bumped specular environment map.
```

Pixel Shader Lighting

The following diagram illustrates the mechanics behind pixel shader lighting.

For example, in the case of a 3-D bump environment map, you sample the normal map and iterate a 3×3 matrix across the polygon using texture coordinates as rows. Then, you can transform and index into a diffuse radiance cube map, or reflect and index into a specular environment cube map. You apply to the base from a previous pass.

Pixel Shader Scope

The pixel shader completely replaces the pixel blending functionality specified by the Microsoft® DirectX® 6.0 and 7.0 multitexturing application programming interfaces (APIs), specifically those operations defined by the D3DTSS_COLOROP, D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAOP, D3DTSS_ALPHAARG1, and D3DTSS_ALPHAARG2 texture stage states, and associated arguments and modifiers. When a procedural pixel shader is set, these states are ignored.

Information on pixel shader scope is divided into the following topics.

- Pixel Shaders and Texture Stage States
- Pixel Shaders and Texture Sampling
- Post-Shader Pixel Processing

Pixel Shaders and Texture Stage States

[C++]
When pixel shaders are in operation, the following texture stage states are still observed.

- D3DTSS_ADDRESSU
- D3DTSS_ADDRESSV
- D3DTSS_ADDRESSW
- D3DTSS_BUMPENVMAT00
- D3DTSS_BUMPENVMAT01
- D3DTSS_BUMPENVMAT10
- D3DTSS_BUMPENVMAT11
- D3DTSS_BORDERCOLOR
- D3DTSS_MAGFILTER
- D3DTSS_MINFILTER

- D3DTSS_MIPFILTER
- D3DTSS_MIPMAPLODBIAS
- D3DTSS_MAXMIPLEVEL
- D3DTSS_MAXANISOTROPY
- D3DTSS_BUMPENVLSCALE
- D3DTSS_BUMPENVLOFFSET
- D3DTSS_TEXCOORDINDEX
- D3DTSS_TEXTURETRANSFORMFLAGS

[Visual Basic]
When pixel shaders are in operation, the following texture stage states are still observed.

- D3DTSS_ADDRESSU
- D3DTSS_ADDRESSV
- D3DTSS_ADDRESSW
- D3DTSS_BUMPENVMAT00
- D3DTSS_BUMPENVMAT01
- D3DTSS_BUMPENVMAT10
- D3DTSS_BUMPENVMAT11
- D3DTSS_BORDERCOLOR
- D3DTSS_MAGFILTER
- D3DTSS_MINFILTER
- D3DTSS_MIPFILTER
- D3DTSS_MIPMAPLODBIAS
- D3DTSS_MAXMIPLEVEL
- D3DTSS_MAXANISOTROPY
- D3DTSS_BUMPENVLSCALE
- D3DTSS_BUMPENVLOFFSET
- D3DTSS_TEXCOORDINDEX
- D3DTSS_TEXTURETRANSFORMFLAGS

Because these texture stage states are not part of the pixel shader, they are not available at shader compile time so the driver can make no assumptions about them. For example, the driver cannot differentiate between bilinear and trilinear filtering at that time. The application is free to change these states without requiring the regeneration of the currently bound shader.

[C++]
Furthermore, calls to the **IDirect3DDevice8::SetTexture** method to change the image data used cannot cause the pixel shader to become invalid, so texture format, in terms of bit depth, is free to change. Calling **SetTexture** with a two-channel texture format such as D3DFMT_A8L8 or D3DFMT_V8U8 must continue to work as though the third channel was set to 0.

[Visual Basic]
Furthermore, calls to the **Direct3DDevice8.SetTexture** method to change the image data used cannot cause the pixel shader to become invalid, so texture format, in terms of bit depth, is free to change. Calling **SetTexture** with a two-channel texture format such as D3DFMT_A8L8 or D3DFMT_V8U8 must continue to work as though the third channel was set to 0.

Pixel Shaders and Texture Sampling

Texture sampling and filtering operations are controlled by the standard texture stage states for minification, magnification, mip filtering, and the wrap addressing modes. For more information, see Texture Stage States.

This information is not available to the driver at shader compile time, so shaders must be able to continue operation when this state changes. The texture type—texture, cube texture, or volume texture—is specified explicitly in the shader syntax. The application is responsible for setting only textures of the correct type during shader execution. Setting a texture of the incorrect type during shader execution will produce unexpected results.

Post-Shader Pixel Processing

In Microsoft® DirectX® 8.0, pixel processing such as fog blending, stencil operations, and render target blending occur after execution of the shader. Render target blending syntax is updated to support new features as described in this topic.

Pixel Shader Inputs

In DirectX 8.0, all colors—diffuse and specular—declared in the texture registers are assumed saturated (clamped) to the range 0 through 1 before use by the shader because this is the range of valid inputs to the shader. Position is assumed to be clipped before the execution of the pixel shader.

All iterated values are assumed by the pixel shader to be iterated-perspective correct, but this is not guaranteed in all hardware. Colors generated as texture coordinates by the address shader are always iterated in a perspective correct manner. However, they are also clamped to the range 0 to 1 during iteration.

Pixel Shader Outputs

The result emitted by the pixel shader is the contents of register r0. Whatever it contains when the shader completes processing is sent to the fog stage and render target blender.

Pixel Shader Data Flow

The section explains the data flow as it is routed to and from the pixel shader virtual machine. The following topics are intended to provide a high-level understanding of developing pixel shaders and the underlying virtual machine.

- Understanding the Pixel Shader Virtual Machine
- Processing Pixel Data

Understanding the Pixel Shader Virtual Machine

The following diagram illustrates the logical flow of data through the pixel shader virtual machine.

[C++]
The data that is interpolated per triangle is routed to the virtual machine as shown in the above diagram. The texture coordinates and color values are interpolated for each pixel. Note that the mapping of texture coordinates is controlled by setting the **D3DTSS_TEXCOORDINDEX** texture stage state. For example, the oD0 data output by the vertex shader would be iterated across a triangle, with the results at each individual pixel being set into the v0 register of the pixel shader.

[Visual Basic]
The data that is interpolated per triangle is routed to the virtual machine as shown in the above diagram. The texture coordinates and color values are interpolated for each pixel. Note that the mapping of texture coordinates is controlled by setting the **D3DTSS_TEXCOORDINDEX** texture stage state. For example, the oD0 data output by the vertex shader would be iterated across a triangle, with the results at each individual pixel being set into the v0 register of the pixel shader.

Processing Pixel Data

For the purpose of understanding how pixel data is processed by the virtual machine, imagine an orange sphere that you want to bump map and place into an environment where lights will shine on its surface. For this simple scenario, assume that the orange color is interpolated using the triangles that form the sphere. The orange color is

interpolated from the data passed down the oD0 channel of the vertex shader, and as such, each pixel is presented to the pixel shader through the v0 register.

The following topics trace the steps needed to render the final image of the orange sphere with the bump and environment map data.

- Preparing Data for the Pixel Shader Virtual Machine
- Sending Data Through the Pixel Shader Virtual Machine
- Pixel Shader Pipeline Order of Operations

Although this section uses a specific example to illustrate how the pixel shader virtual machine processes data, the principles and steps demonstrated are valid for developing a variety of pixel shaders.

Preparing Data for the Pixel Shader Virtual Machine

[C++]
Before rendering the orange sphere with the bump and environment map data, you must load the bump and environment maps. Then set the relevant texture stage states for each of the texture operation units. This includes setting the D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT01, D3DTSS_BUMPENVMAT10, D3DTSS_BUMPENVMAT11, and D3DTSS_TEXCOORDINDEX texture stage states. However, the color operations—defined by D3DTSS_COLOROP—do not need to be set. This is because the pixel shader code is used to define the operations performed on a per-pixel basis. In general, the texture stage states that remain constant while rendering a triangle, object, or frame are set before invoking the pixel shader virtual machine. This includes the 2×2 bump-mapping matrix values. However, the data that is used to shade each pixel is specified through the pixel shader language.

[Visual Basic]
Before rendering the orange sphere with the bump and environment map data, you must load the bump and environment maps. Then set the relevant texture stage states for each of the texture operation units. This includes setting the D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT01, D3DTSS_BUMPENVMAT10, D3DTSS_BUMPENVMAT11, and D3DTSS_TEXCOORDINDEX texture stage states. However, the color operations—defined by D3DTSS_COLOROP—do not need to be set. This is because the pixel shader code is used to define the operations performed on a per-pixel basis. In general, the texture stage states that remains constant while rendering a triangle, object, or frame are set before invoking the pixel shader virtual machine. This includes the 2×2 bump-mapping matrix values. However, the data that is used to shade each pixel is specified through the pixel shader language.

For this example, the front end processing that occurs in the vertex shader is not addressed. Instead, assume that the front end sets up the texture interpolation so that the bump data is passed through the oT0 channel of the vertex shader. For the orange sphere example, this means that the bump map address used to access the bump data is set to the first texture coordinate register of the pixel shader (t0). In a similar fashion, the light map address—before the modification done in the texture operation unit—is passed from the vertex shader oT1 channel to the pixel shader through t1.

Sending Data Through the Pixel Shader Virtual Machine

The address used for the texture operation for stage 0 is the bump map addresses presented through the first texture coordinate register (t0). The operation is a simple look-up of the bump data, as illustrated by the following pixel shader code.

```
tex t0
```

The texture operation for stage 1 is more complicated. In this case, you must set the texture stage state so that the 2×2 matrix is loaded. Note that there is a 2×2 matrix that can be loaded to simulate the change between the eye-point and the light source. This enables the bump and environment map to be reused as the eye-point is moved around a scene. Therefore, the address into the environment map is a base address, presented through the second texture coordinate, t1, modulated by the eye-point adjustment and local bump value. More precisely, the u coordinate into the environment map is u = t1.r + mat00 ∗ t0.r + mat01 ∗ t0.g, and the v coordinate into the environment map is v = t1.g + mat10 ∗ t0.r + mat11 ∗ t0.g. The values in the t0.r and t0.g register components are actually the du and dv values that represent the bump data looked-up in stage 0 and stored in register t0.

Note

The u and v address calculations can require two slots in the actual pixel shader code. This is because most hardware reuses the logic for the u and then the v address calculation. Current hardware does not provide for the addresses to be computed in parallel.

The following example shows how this is done in pixel shader code.

```
ps.1.0        // DirectX8 Version.
tex t0        // Declare the texture (the bump data).
texbem t1, t0  // Declare the environment map and get data from t0.
mov r0, v0     // Note the assumption that the diffuse orange color
               // is fed into the pixel shader through register v0 for
               // each pixel.
add r0, r0, t1 // Add the specular environment data.
```

However, this simple example is not the most common case. Typically, there is a base texture applied to the object before performing the bump environment mapping. For example, you might send a white sphere that is used as a base for applying the texture data. This is the diffuse light that is brighter on part of the surface and less bright on other parts. In this case, you move the bump environment mapping operations out a

stage and modulate the base texture by the diffuse light. The following example shows how this is done in pixel shader code.

```
ps.1.0        // DirectX8 Version.
tex t0        // Declare the base texture. It could be something
              // like marble or wood or any other surface material.
tex t1        // Declare the texture (the bump data).
texbem t2, t1  // Declare the environment map.
mul r0, v0, t0 // Modulate the base texture by the diffuse light.
add r0, r0, t2 // Add the specular environment data.
```

You can use traditional texture mapping (one stage), interpolate the specular, and have it available through register v1 of the pixel shader virtual machine. Other complex pixel shaders can be constructed using a set of fixed constants stored in the constant registers (c0-c7). Also, the texture registers are often used with texture stages but are read/write to the pixel arithmetic logic unit (ALU). Thus the pixel shader can use a texture register as a temporary register. Note also that r0 and r1 are read/write to the ALU, but the results stored in r0 are the only results that are sent to the depth and stencil logic. In fact, the only way that results can be passed downstream is through the r0 register.

Pixel Shader Pipeline Order of Operations

[C++]
You can control the states of the pixel shader pipeline throughout its execution by setting render states and texture stage states, and by calling the **IDirect3DDevice8::SetTexture**, **IDirect3DDevice8::SetTextureStageState**, **IDirect3DDevice8::SetPixelShader**, and **IDirect3Ddevice8::SetPixelShaderConstant** methods. These states and functions specify the flow of control in the pixel shader and the states of the steps in the pixel shader pipeline. The following diagram illustrates the order of these steps and the settings that most directly control them.

[Visual Basic]
You can control the states of the pixel shader pipeline throughout its execution by setting render states and texture stage states, and by calling the **Direct3DDevice8.SetTexture**, **Direct3DDevice8.SetTextureStageState**, **Direct3DDevice8.SetPixelShader**, and **Direct3Ddevice8.SetPixelShaderConstant** methods. These states and functions specify the flow of control in the pixel shader and the states of the steps in the pixel shader pipeline. The following diagram illustrates the order of these steps and the settings that most directly control them.

Depending on the value of the handle passed to **SetPixelShader**, either the fixed function or pixel shader pipeline is used in any one instance, but not both. The depth stencil test may occur prior to the time indicated by the figure, but this does not affect the results of the color pipeline. When stepping through a shading algorithm, keep in mind the various pipeline operations that can clamp values.

Using Pixel Shaders

This section shows you how to use pixel shaders in your application.

- Determining Pixel Shader Support
- Setting Pixel Shader Texture Inputs
- Setting the Pixel Shader Constant Registers
- Compiling a Pixel Shader
- Creating a Pixel Shader

Determining Pixel Shader Support

[C++]
You can query members of **D3DCAPS8** to determine the level of support for operations involving pixel shaders. The following table lists the device capabilities related to programmable pixel processing in Microsoft® DirectX® 8.0.

Device capability	Description
MaxPixelShaderValue	Maximum value of pixel shader arithmetic component.
MaxSimultaneousTextures	Number of texture declaration instructions supported.
MaxTextureBlendStages	Number of instructions supported.
PixelShaderVersion	Level of support for pixel shaders.

The **MaxSimultaneousTextures** and **MaxTextureBlendStages** capabilities only apply to fixed-function pixel blending(using multi-texture).

The **PixelShaderVersion** capability indicates the level of pixel shader supported. Only pixel shaders with version numbers equal to or less than this value will be successfully created when calling the method **IDirect3DDevice8::CreatePixelShader**. The major version number is encoded in the second byte of **PixelShaderVersion**. The low byte contains a minor version number. The pixel shader version is indicated by the first token in each shader.

[Visual Basic]

You can query members of **D3DCAPS8** to determine the level of support for operations involving pixel shaders. The following table lists the device capabilities related to programmable pixel processing in Microsoft® DirectX® 8.0.

Device capability	Description
MaxPixelShaderValue	Maximum value of pixel shader arithmetic component.
MaxSimultaneousTextures	Number of texture declaration instructions supported.
MaxTextureBlendStages	Number of instructions supported.
PixelShaderVersion	Level of support for pixel shaders.

The **MaxSimultaneousTextures** and **MaxTextureBlendStages** capabilities only apply to fixed-function pixel blending(using multi-texture).

The **PixelShaderVersion** capability indicates the level of pixel shader supported. Only pixel shaders with version numbers equal to or less than this value will be successfully created when calling the method **Direct3DDevice8.CreatePixelShader**. The major version number is encoded in the second byte of **PixelShaderVersion**. The low byte contains a minor version number. The pixel shader version is indicated by the first token in each shader.

Pixel shader applications do not need to check the **MaxSimultaneousTextures** and **MaxTextureBlendStages** device capabilities.

Each implementation sets the **PixelShaderVersion** member to indicate the maximum pixel shader version that it can fully support. This implies that implementations should never fail the creation of a valid shader of the version less than or equal to the version reported by **PixelShaderVersion**.

The following table summarizes the supported pixel shader versions and the associated level of functionality and register counts.

Version	Functionality	tn	rn	cn	vn
1.0	DirectX 8.0	4	2	8	2
1.1	DirectX 8.0	4	2	8	2

Version 1.0 supports all the operations documented in the Instructions reference topic for pixel shaders. A maximum of four texture address operations and a total of eight instructions are supported. In this version the texture registers ***tn*** are read-only, and only one can be used in a given instruction.

Version 1.1 supports instructions, with limits of four address operations and eight blending operations. In this version the texture registers are read/write, and two can be used as inputs in a single instruction.

Setting Pixel Shader Texture Inputs

[C++]
There is a direct mapping between the input register colors and the texture stages. The temporary registers of the Direct3DX pixel shader assembler are preloaded with the texture colors from the corresponding texture stages. Conversely, the textures are bound directly to the temporary registers by calling the **IDirect3DDevice8::SetTexture** method.

[Visual Basic]
There is a direct mapping between the input register colors and the texture stages. The temporary registers of the Direct3DX pixel shader assembler are preloaded with the texture colors from the corresponding texture stages. Conversely, the textures are bound directly to the temporary registers by calling the **Direct3DDevice8.SetTexture** method.

In addition, texture coordinates are mapped directly and defined by texture declarations.

Setting the Pixel Shader Constant Registers

[C++]
You can use the following methods to set and retrieve the values in the pixel shader constant registers.

- **IDirect3DDevice8::GetPixelShaderConstant**
- **IDirect3DDevice8::SetPixelShaderConstant**

In addition, you can use the **def** Direct3DX assembler instruction to set the constant registers of a pixel shader.

[Visual Basic]
You can use the following methods to set and retrieve the values in the pixel shader constant registers.

- **Direct3DDevice8.GetPixelShaderConstant**
- **Direct3DDevice8.SetPixelShaderConstant**

In addition, you can use the **def** Direct3DX assembler instruction to set the constant registers of a pixel shader.

Compiling a Pixel Shader

[C++]
The Direct3DX utility library provides a set of functions to compile pixel shaders. The following functions are provided.

- **D3DXAssembleShader**
- **D3DXAssembleShaderFromFileA**
- **D3DXAssembleShaderFromFileW**

[Visual Basic]
The Direct3DX utility library provides a set of functions to compile pixel shaders. The following methods are provided.

- **D3DX8.AssembleShader**
- **D3DX8.AssembleShaderFromFile**

Creating a Pixel Shader

[C++]
The **IDirect3DDevice8::CreatePixelShader** method is used to create a pixel shader in Microsoft® DirectX® 8.0. You pass a declaration that defines the blending operations. Microsoft Direct3D® returns a shader handle. The shader validation is done when **CreatePixelShader** is called.

[Visual Basic]
The **Direct3DDevice8.CreatePixelShader** method is used to create a pixel shader in Microsoft® DirectX® 8.0. You pass a declaration that defines the blending operations. Microsoft Direct3D® returns a shader handle. The shader validation is done when **CreatePixelShader** is called.

Note that a given shader may fail creation due to the restraints of the DirectX 8.0 hardware model.

Pixel Shader Examples

This section provides a list of common effects implemented with pixel shader code. Information is divided into the following topics.

- Diffuse Lighting
- Specular Mapping

- Diffuse Lighting and Specular Mapping
- Dot Product Lighting
- Pixel Shader Sample

Diffuse Lighting

The following shader performs per-pixel diffuse lighting. Use the **dp3** instruction to compute diffuse intensity.

```
ps.1.0                  // DirectX8 Version.
tex t0                  // Declare the n-map.
texm3x3pad t1, t0_bx2   // First row of transform.
texm3x3pad t2, t0_bx2   // Second row of transform.
texm3x3tex t3, t0_bx2   // Third row of transform.
dp3_sat r0, t3_bx2, v0_bx2
```

Specular Mapping

The following shader performs per-pixel specular mapping.

```
ps.1.0              // DirectX8 Version.
tex t0              // Declare the n-map.
texm3x3pad   t1, t0     // First row of transform.
texm3x3pad   t2, t0     // Second row of transform.
texm3x3spec  t3, t0, c0  // Third row of transform. Then reflect
                // infinite and sample the cube map.

mov r0, t3          // Emit to the frame buffer blender.
                // This adds to the frame buffer contents.
```

Diffuse Lighting and Specular Mapping

The following combination shader performs per-pixel diffuse lighting and specular mapping.

```
ps.1.0              // DirectX8 Version.
tex t0              // Declare the n-map.
texm3x3pad  t1, t0    // First row of transform.
texm3x3pad  t2, t0    // Second row of transform.
texm3x3spec t3, t0    // Third row of transform. Sample diffuse cube
                // map to t2 and specular cube map into t3.
mov r0, t3          // Output the specular color.
```

Dot Product Lighting

Dot products require signed data, so you need to convert the inputs using the **_bx2** argument modifier. Lighting clamps negative values using the **_sat** instruction modifier.

```
// Diffuse is the light direction.

ps.1.0      // DirectX8 Version
tex t0      // The normal map
dp3_sat r0, t0_bx2, v0_bx2
```

Pixel Shader Sample

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The MFCPixelShader Sample shows the effect that different pixel shaders have on a rendered scene. This sample has six predefined shaders to select from, it and enables the user to modify the predefined pixel shaders to get different results.

Point Sprites

This section discusses point sprites. Information is divided into the following topics.

- What Are Point Sprites?
- Point Primitive Rendering Controls
- Point Size Computations
- Point Rendering

What Are Point Sprites?

Support for point sprites in Microsoft® Direct3D® for Microsoft DirectX® 8.0 enables the high-performance rendering of points (particle systems). Point sprites are generalizations of generic points that enable arbitrary shapes to be rendered as defined by textures.

Point sprites are not supported in the fixed-function geometry pipeline. To use point sprites, programmable vertex processing must be used.

Point Primitive Rendering Controls

[C++]
Microsoft® Direct3D® for Microsoft DirectX® 8.0 supports additional parameters to control the rendering of point sprites (point primitives). These parameters enable points to be of variable size and have a full texture map applied. The size of each point is determined by an application-specified size combined with a distance-based function computed by Direct3D. The application can specify point size either as per-vertex or by setting D3DRS_POINTSIZE, which applies to points without a per-vertex size. The point size is expressed in camera space units, with the exception of when the application is passing post-transformed flexible vertex format (FVF) vertices. In this case, the distance-based function is not applied and the point size is expressed in units of pixels on the render target.

The texture coordinates computed and used when rendering points depends on the setting of D3DRS_POINTSPRITEENABLE. When this value is set to TRUE, the texture coordinates are set so that each point displays the full-texture. In general, this is only useful when points are significantly larger than one pixel. When D3DRS_POINTSPRITEENABLE is set to FALSE, each point's vertex texture coordinate is used for the entire point.

[Visual Basic]
Microsoft® Direct3D® for Microsoft DirectX® 8.0 supports additional parameters to control the rendering of point sprites (point primitives). These parameters enable points to be of variable size and have a full-texture map applied. The size of each point is determined by an application-specified size combined with a distance-based function computed by Direct3D. The application can specify point size either as per-vertex or by setting D3DRS_POINTSIZE, which applies to points without a per-vertex size. The point size is expressed in camera space units, with the exception of when the application is passing post-transformed flexible vertex format (FVF) vertices. In this case, the distance-based function is not applied and the point size is expressed in units of pixels on the render target.

The texture coordinates computed and used when rendering points depends on the setting of D3DRS_POINTSPRITEENABLE. When this value is set to TRUE, the texture coordinates are set so that each point displays the full texture. In general, this is only useful when points are significantly larger than one pixel. When D3DRS_POINTSPRITEENABLE is set to FALSE, each point's vertex texture coordinate is used for the entire point.

Point Size Computations

[C++]

Point size is determined by D3DRS_POINTSCALEENABLE. If this value is set to FALSE, the application-specified point size is used as the screen-space (post-transformed) size. Vertices that are passed to Microsoft® Direct3D® in screen space do not have point sizes computed; the specified point size is interpreted as screen-space size.

If D3DRS_POINTSCALEENABLE is TRUE, Direct3D computes the screen-space point size according to the following formula. The application-specified point size is expressed in camera space units.

$$S_s = V_h * S_i * \text{sqrt}(1/(A + B*D_e + C*(D_e^2)))$$

In this formula, the input point size, S_i, is either per-vertex or the value of the D3DRS_POINTSIZE render state. The point scale factors, D3DRS_POINTSCALE_A, D3DRS_POINTSCALE_B, and D3DRS_POINTSCALE_C, are represented by the points A, B, and C. The height of the viewport, V_h, is the **Height** member of the **D3DVIEWPORT8** structure representing the viewport. D_e, the distance from the eye to the position (the eye at the origin), is computed by taking the eye space position of the point (X_e, Y_e, Z_e) and performing the following operation.

$$D_e = \text{sqrt}\ (X_e^2 + Y_e^2 + Z_e^2)$$

The maximum point size, P_{max}, is determined by taking the smaller of either the **MaxPointSize** member of **D3DCAPS8** or the D3DRS_POINTSIZE_MAX render state. The minimum point size, P_{min}, is determined by querying the value of D3DRS_POINTSIZE_MIN. Thus the final screen-space point size, S, is determined in the following manner.

- If $S_s > P_{max}$, then $S = P_{max}$
- if $S_s < P_{min}$, then $S = P_{min}$
- Otherwise, $S = S_s$

[Visual Basic]
Point size is determined by D3DRS_POINTSCALEENABLE. If this value is set to False, the application-specified point size is used as the screen-space (post-transformed) size. Vertices that are passed to Microsoft® Direct3D® in screen space do not have point sizes computed; the specified point size is interpreted as screen-space size.

If D3DRS_POINTSCALEENABLE is True, Direct3D computes the screen-space point size according to the following formula. The application-specified point size is expressed in camera space units.

$$S_s = V_h * S_i * \text{sqrt}(1/(A + B*D_e + C*(D_e^2)))$$

In this formula, the input point size, S_i, is either per-vertex or the value of the D3DRS_POINTSIZE render state. The point scale factors, D3DRS_POINTSCALE_A, D3DRS_POINTSCALE_B, and

D3DRS_POINTSCALE_C, are represented by the points ***A***, ***B***, and ***C***. The height of the viewport, V_h, is the **Height** member of the **D3DVIEWPORT8** type representing the viewport. D_e, the distance from the eye to the position (the eye at the origin), is computed by taking the eye space position of the point (X_e, Y_e, Z_e) and performing the following operation.

$D_e = \text{sqrt}\ (X_e^2 + Y_e^2 + Z_e^2)$

The maximum point size, P_{max}, is determined by taking the smaller of either the **MaxPointSize** member of **D3DCAPS8** or the D3DRS_POINTSIZE_MAX render state. The minimum point size, P_{min}, is determined by querying the value of D3DRS_POINTSIZE_MIN. Thus the final screen-space point size, S, is determined in the following manner.

- If $S_s > P_{max}$, then $S = P_{max}$
- if $S_s < P_{min}$, then $S = P_{min}$
- Otherwise, $S = S_s$

Point Rendering

[C++]
A screen-space point,

P = (**X**, **Y**, **Z**, **W**)

of screen-space size **S** is rasterized as a quadrilateral of the following four vertices.

((**X**+**S**/2, **Y**+**S**/2, **Z**, **W**), (**X**+**S**/2, **Y**-**S**/2, **Z**, **W**), (**X**-**S**/2, **Y**-**S**/2, **Z**, **W**), (**X**-**S**/2, **Y**+**S**/2, **Z**, **W**))

The vertex color attributes are duplicated at each vertex; thus each point is always rendered with constant colors.

The assignment of texture indices is controlled by the D3DRS_POINTSPRITEENABLE render state setting. If D3DRS_POINTSPRITEENABLE is set to FALSE, then the vertex texture coordinates are duplicated at each vertex. If D3DRS_POINTSPRITEENABLE is set to TRUE, then the texture coordinates at the four vertices are set to the following values.

(0.F, 0.F), (0.F, 1.F), (1.F, 0.F), (1.F, 1.F)

This is shown in the following diagram.

When clipping is enabled, points are clipped in the following manner. If the vertex exceeds the range of depth values—**MinZ** and **MaxZ** of the **D3DVIEWPORT8**

structure—into which a scene is to be rendered, the point exists outside of the view frustum and is not rendered. If the point, taking into account the point size, is completely outside the viewport in **X** and **Y**, then the point is not rendered; the remaining points are rendered. It is possible for the point position to be outside the viewport in **X** or **Y** and still be partially visible.

Points may or may not be correctly clipped to user-defined clip planes. If D3DPMISCCAPS_CLIPPLANESCALEDPOINTS is not set in the **PrimitiveMiscCaps** member of **D3DCAPS8**, then points are clipped to user-defined clip planes based only on the vertex position, ignoring the point size. In this case, scaled points are fully rendered when the vertex position is inside the clip planes, and discarded when the vertex position is outside a clip plane. Applications can prevent potential artifacts by adding a border geometry to clip planes that is as large as the maximum point size.

If the D3DPMISCCAPS_CLIPPLANESCALEDPOINTS bit is set, then the scaled points are correctly clipped to user-defined clip planes.

Hardware vertex processing may or may not support point size. For example, if a device is created with D3DCREATE_HARDWARE_VERTEXPROCESSING on a HAL Device (D3DDEVTYPE_HAL) that has the **MaxPointSize** member of its **D3DCAPS8** structure set to 1.0 or 0.0, then all points are a single pixel. To render pixel point sprites less than 1.0, you must use either flexible vertex format (FVF) TL (transformed and lit) vertices or software vertex processing (D3DCREATE_SOFTWARE_VERTEXPROCESSING), in which case the Microsoft® Direct3D® run time emulates the point sprite rendering. For details on FVF TL vertices, see Transformed and Lit Vertex Functionality.

A hardware device that does vertex processing and supports point sprites—**MaxPointSize** set to greater than 1.0f—is required to perform the size computation for nontransformed sprites and is required to properly set the per-vertex or D3DRS_POINTSIZE for TL vertices.

[Visual Basic]
A screen-space point,

P = (**X**, **Y**, **Z**, **W**)

of screen-space size **S** is rasterized as a quadrilateral of the following four vertices.

((**X**+**S**/2, **Y**+**S**/2, **Z**, **W**), (**X**+**S**/2, **Y**-**S**/2, **Z**, **W**), (**X**-**S**/2, **Y**-**S**/2, **Z**, **W**), (**X**-**S**/2, **Y**+**S**/2, **Z**, **W**))

The vertex color attributes are duplicated at each vertex; thus each point is always rendered with constant colors.

The assignment of texture indices is controlled by the D3DRS_POINTSPRITEENABLE render state setting. If D3DRS_POINTSPRITEENABLE is set to False, then the vertex texture coordinates

are duplicated at each vertex. If D3DRS_POINTSPRITEENABLE is set to True, then the texture coordinates at the four vertices are set to the following values.

(0.0, 0.0), (0.0, 1.0), (1.0, 0.0), (1.0, 1.0)

This is shown in the following diagram.

When clipping is enabled, points are clipped in the following manner. If the vertex exceeds the range of depth values—**MinZ** and **MaxZ** of the **D3DVIEWPORT8** type—into which a scene is to be rendered, the point exists outside of the view frustum and is not rendered. If the point, taking into account the point size, is completely outside the viewport in **x** and **y**, then the point is not rendered; the remaining points are rendered. It is possible for the point position to be outside the viewport in **x** or **y** and still be partially visible.

Points may or may not be correctly clipped to user-defined clip planes. If the D3DPMISCCAPS_CLIPPLANESCALEDPOINTS is not set in the **PrimitiveMiscCaps** member of **D3DCAPS8**, then points are clipped to user-defined clip planes based only on the vertex position, ignoring the point size. In this case, scaled points are fully rendered when the vertex position is inside the clip planes, and discarded when the vertex position is outside a clip plane. Applications can prevent potential artifacts by adding a border geometry to clip planes that is as large as the maximum point size.

If the D3DPMISCCAPS_CLIPPLANESCALEDPOINTS bit is set, then the scaled points are correctly clipped to user-defined clip planes.

Hardware vertex processing may or may not support point size. For example, if you create a device with D3DCREATE_HARDWARE_VERTEXPROCESSING on a HAL Device (D3DDEVTYPE_HAL) that has the **MaxPointSize** member of its **D3DCAPS8** type set to 1.0 or 0.0, then all points are a single pixel. To render pixel point sprites less than 1.0, you must use either flexible vertex format (FVF) TL (transformed and lit) vertices or software vertex processing (D3DCREATE_SOFTWARE_VERTEXPROCESSING), in which case the Microsoft® Direct3D® run time emulates the point sprite rendering. For details on FVF TL vertices, see Transformed and Lit Vertex Functionality.

A hardware device that does vertex processing and supports point sprites—**MaxPointSize** set to greater than 1.0f—is required to perform the size computation for nontransformed sprites and is required to properly set the per-vertex or D3DRS_POINTSIZE for TL vertices.

Vertex Shaders

This section describes vertex shaders and how they are used in Microsoft® Direct3D® applications. The following topics are discussed.

- Introduction to Vertex Shaders
- Vertex Shader Components
- Using Vertex Shaders
- Vertex Shader Techniques and Applications
- Vertex Shader Examples

All vertex shader code and syntax used in this documentation is based on and compatible with the Direct3DX vertex shader assembler. Before reading this section, ensure that you are familiar with the vertex shader assembler by studying the Vertex Shader Assembler Reference.

Introduction to Vertex Shaders

Vertex shaders control the loading and processing of vertices. Microsoft® Direct3D® for Microsoft DirectX® 8.0 supports two types of vertex processing: programmable vertex processing and fixed-function vertex processing. In DirectX 8.0 documentation, the term *vertex shader* most often applies to the programmable mode, although the vertex shader API mechanism encompasses both types of functionality. Vertex shaders are defined at creation as using programmable vertex processing or fixed-function vertex processing. Once created, a vertex shader is referred to by its handle. Vertex shaders are not editable. To change a vertex shader, it must be destroyed and recreated.

Each vertex shader includes a function, which defines the operations to apply to each vertex, and a declaration, which defines the inputs to the shader, including how vertex elements in the input data streams are used by the shader.

Vertex processing performed by vertex shaders encompasses only operations applied to single vertices. The output of the vertex processing step is defined as individual vertices, each of which consists of a clip-space position (x, y, z, and w) plus color, texture coordinate, fog intensity, and point size information. The projection and mapping of these positions to the viewport, the assembling of multiple vertices into primitives, and the clipping of primitives is done by a subsequent processing stage and is not under the control of the vertex shader.

[C++]
Fixed function vertex processing provides the same functionality as in Direct3D for DirectX 7.0, including transformation and lighting, vertex blending, and texture coordinate generation. Unlike programmed vertex shaders, where the operations applied to vertices are defined within the shader, fixed-function vertex processing is controlled by a device state set by methods on **IDirect3DDevice8**, which set lights, transforms, and so on. It is still useful to have multiple fixed-function vertex shaders,

because even though the function is fixed and shared, the declarations can vary. This enables feeding the fixed-function shader with differing layouts of multistream inputs.

[Visual Basic]
Fixed function vertex processing provides the same functionality as in Direct3D for DirectX 7.0, including transformation and lighting, vertex blending, and texture coordinate generation. Unlike programmed vertex shaders, where the operations applied to vertices are defined within the shader, fixed-function vertex processing is controlled by a device state set by methods on **Direct3DDevice8**, which set lights, transforms, and so on. It is still useful to have multiple fixed-function vertex shaders because even though the function is fixed and shared, the declarations can vary. This enables feeding the fixed-function shader with differing layouts of multistream inputs.

The input vertex elements for fixed-function vertex processing have fixed semantics. Thus the declaration tags specify input vertex elements as coordinate, normal, color, and so on. The vertex outputs for fixed-function vertex shaders always include coordinates, diffuse and specular colors, and multiple texture coordinates, as required by the current device state settings for pixel processing.

Programmed vertex shaders have a function defined by an array of instructions to apply to each vertex. The mapping of the input vertex elements to the vertex input registers for programmed vertex shaders is defined in the shader declaration, but the input vertex elements do not have specific semantics about their use. The interpretation of the input vertex elements is up to the shader instructions. The vertex outputs for programmed vertex shaders are explicitly written to by instructions in the shader function.

For more information on the Direct3DX vertex shader assembler, see Vertex Shader Assembler Reference.

Vertex Shader Components

This section describes the components of a vertex shader. This information is divided into the following topics.

- Vertex Shader Declaration
- Vertex Shader Function

Vertex Shader Declaration

[C++]
The declaration portion of a vertex shader defines the static external interface of the shader. A vertex shader declaration includes the following information.

- Binding stream data to vertex shader input registers. This information defines the type and vertex input register assignment of each element in each data stream. The type specifies the arithmetic data type and the dimensionality—one, two, three, or four values. Stream data elements that are less than four values are always expanded to four values with zero or more 0.F values and one 1.F value.
- Binding vertex shader input registers to implicit data from the primitive tessellator. This controls the loading of vertex data that is not loaded from a stream but rather is generated during primitive tessellation prior to the vertex shader.
- Loading data in the constant memory at the time that a shader is set as the current shader. Each token specifies values for one or more contiguous 4-**DWORD** constant registers. This enables the shader to update an arbitrary subset of the constant memory, overwriting the device state, which contains the current values of the constant memory. These values can be overwritten, between DrawPrimitive calls, during the time that a shader is bound to a device through the **IDirect3DDevice8::SetVertexShaderConstant** method.

Declaration arrays are single-dimensional arrays of **DWORD**s composed of multiple tokens, each of which is one or more **DWORD**s. The single-**DWORD** token value 0xFFFFFFFF is a special token used to indicate the end of the declaration array. The single **DWORD** token value 0x00000000 is a padding token that is ignored during the declaration parsing. This pad token enables more complex in-place editing of declarations. Note that 0x00000000 is a valid value for **DWORD**s following the first **DWORD** for multiple word tokens.

The following code example is an example of writing declarators. It shows how to define a vertex using a simple structure.

```
struct Vertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Normal;
    D3DCOLOR    Diffuse;
    D3DXVECTOR2 TexCoord0;
};
```

Now consider the following code example, which shows how to define the same vertex structure above as a declarator.

```
DWORD dwDecl[] =
{
    D3DVSD_STREAM( 0 ),
    D3DVSD_REG( D3DVSDE_POSITION,  D3DVSDT_FLOAT3 ),
    D3DVSD_REG( D3DVSDE_NORMAL,    D3DVSDT_FLOAT3 ),
    D3DVSD_REG( D3DVSDE_DIFFUSE,   D3DVSDT_D3DCOLOR ),
    D3DVSD_REG( D3DVSDE_TEXCOORD0, D3DVSDT_FLOAT2 ),
    D3DVSD_END()
```

```
};

DWORD dwFvf = D3DFVF_POSITION | D3DFVF_NORMAL | D3DFVF_DIFFUSE |
        D3DFVF_TEX0 | D3DFVF_TEXCOORDSIZE2(0);
```

The declarator sets data stream 0, defines a position and normal vector composed of three float values, defines a diffuse component composed of a four component unsigned byte, and defines a pair of texture coordinates composed of two float values.

When you use the fixed-function vertex shader, the mapping of the vertex input registers is fixed so that specific vertex elements, such as position or normal, must be placed in specific register locations in the vertex input memory. These assignments are made automatically when passing an FVF code to **IDirect3DDevice8::SetVertexShader**. When you use an explicit shader declaration, the **D3DVSDE** preprocessor macros define the vertex input location into which specific elements must be loaded. See D3d8types.h for a definition of the macros used to generate the declaration token array.

For information on the shader declaration token types and bit definitions, see Vertex Shader Declarator Macros.

Notes

When you use fixed-function vertex shaders, the vertex stride set in **IDirect3DDevice8::SetStreamSource** and vertex size must be the same size. For vertex shaders that use declarations, the stream stride can be greater than the declaration stride.

[Visual Basic]
The declaration portion of a vertex shader defines the static external interface of the shader. A vertex shader declaration includes the following information.

- Binding stream data to vertex shader input registers. This information defines the type and vertex input register assignment of each element in each data stream. The type specifies the arithmetic data type and the dimensionality—one, two, three, or four values. Stream data elements that are less than four values are always expanded to four values with zero or more 0.F values and one 1.F value.
- Binding vertex shader input registers to implicit data from the primitive tessellator. This controls the loading of vertex data that is not loaded from a stream but is generated during primitive tessellation prior to the vertex shader.
- Loading data in the constant memory at the time that a shader is set as the current shader. Each token specifies values for one or more contiguous 4-**DWORD** constant registers. This enables the shader to update an arbitrary subset of the constant memory, overwriting the device state, which contains the current values of the constant memory. These values can be overwritten, between DrawPrimitive calls, during the time that a shader is bound to a device through the **Direct3DDevice8.SetVertexShaderConstant** method.

Declaration arrays are single-dimensional arrays of **Long**s composed of multiple tokens, each of which is one or more **Long**s. The single-**Long** token value 0xFFFFFFFF is a special token used to indicate the end of the declaration array. The single **Long** token value 0x00000000 is a padding token that is ignored during the declaration parsing. This pad token enables more complex in-place editing of declarations. Note that 0x00000000 is a valid value for **Long**s following the first **Long** for multiple word tokens.

The following code example demonstrates how to write declarators. It shows how to define a vertex using a simple structure.

```
Type Vertex
    Position As D3DVECTOR
    Normal As D3DVECTOR
    Diffuse As D3DCOLORVALUE
    TexCoord0 As D3DVECTOR2
End Type
```

Now consider the following code example, which shows how to define the same vertex structure above as a declarator.

```
Dim Decl(5) As Long
Dim FVF as Long

Decl(0) = D3DVSD_STREAM(0)
Decl(1)= D3DVSD_STREAM( 0 )D3DVSD_REG( D3DVSDE_POSITION,
D3DVSDT_FLOAT3 ),
Decl(2)= D3DVSD_STREAM( 0 )D3DVSD_REG( D3DVSDE_NORMAL,
D3DVSDT_FLOAT3 ),
Decl(3)= D3DVSD_STREAM( 0 )D3DVSD_REG( D3DVSDE_DIFFUSE,
D3DVSDT_D3DCOLOR ),
Decl(4)= D3DVSD_STREAM( 0 )D3DVSD_REG( D3DVSDE_TEXCOORD0,
D3DVSDT_FLOAT2 ),
Decl(5)= D3DVSD_STREAM( 0 )D3DVSD_END()

FVF = D3DFVF_POSITION Or D3DFVF_NORMAL Or D3DFVF_DIFFUSE Or _
    D3DFVF_TEX0 Or D3DFVF_TEXCOORDSIZE2(0);
```

The declarator sets data stream 0, defines a position and normal vector composed of three **Long** values, defines a diffuse component composed of a four component unsigned byte, and defines a pair of texture coordinates composed of two **Long** values.

When you use the fixed-function vertex shader, the mapping of the vertex input registers is fixed so that specific vertex elements, such as position or normal, must be placed in specific register locations in the vertex input memory. These assignments are made automatically when passing an FVF code to **Direct3DDevice8.SetVertexShader**. When you use an explicit shader declaration,

the **D3DVSDE** preprocessor macros define the vertex input location into which specific elements must be loaded. See \Samples\Multimedia\VBSamples\Common\D3DShaders.bas for a definition of the macros used to generate the declaration token array.

For information on the shader declaration token types and bit definitions, see Vertex Shader Declarator Functions.

Notes

When using fixed-function vertex shaders, the vertex stride set in **Direct3DDevice8.SetStreamSource** and vertex size must be the same size. For vertex shaders that use declarations, the stream stride can be greater than the declaration stride.

Vertex Shader Function

The function portion of a vertex shader defines the operations applied to each vertex. Only programmed vertex shaders have shader functions. Like declarations, shader functions are single dimensional arrays of **DWORD**s that form an ordered list of instructions to execute for each vertex. Each instruction is composed of multiple **DWORD**s.

The Direct3DX utility library supplies a set of functions to generate the standard **DWORD** tokens transmitted to the driver. For details, see Compiling a Vertex Shader.

Using Vertex Shaders

This section shows you how to define and use vertex shaders in your application and introduces some effects that can be achieved with vertex shaders.

- Determining Vertex Shader Support
- Setting the Vertex Shader Constant Registers
- Compiling a Vertex Shader
- Creating a Vertex Shader

Determining Vertex Shader Support

[C++]
You can query members of **D3DCAPS8** to determine the level of support for operations involving vertex shaders. The following table lists the device capabilities related to programmable vertex processing in Microsoft® DirectX® 8.0.

Device capability	Description
MaxPrimitiveCount	Maximum number of primitives for each

	DrawPrimitive call.
MaxVertexIndex	Maximum size of indices supported for hardware vertex processing.
MaxStreams	Maximum number of concurrent data streams.
MaxStreamStride	Maximum data stream for a stride.
MaxVertexShaderConst	Number of vertex shader constant registers.
VertexShaderVersion	Level of support for vertex shaders.

The **VertexShaderVersion** capability indicates the level of vertex shader supported. Only vertex shaders with version numbers equal to or less than this value will be successfully created when calling the method **IDirect3DDevice8::CreateVertexShader**. The level of vertex shader is specified to **CreateVertexShader** as the first token in the shader token stream.

[Visual Basic]
You can query members of **D3DCAPS8** to determine the level of support for operations involving vertex shaders. The following table lists the device capabilities related to programmable vertex processing in Microsoft® DirectX® 8.0.

Device capability	**Description**
MaxPrimitiveCount	Maximum number of primitives for each DrawPrimitive call.
MaxVertexIndex	Maximum size of indices supported for hardware vertex processing.
MaxStreams	Maximum number of concurrent data streams.
MaxStreamStride	Maximum data stream for a stride.
MaxVertexShaderConst	Number of vertex shader constant registers.
VertexShaderVersion	Level of support for vertex shaders.

The **VertexShaderVersion** capability indicates the level of vertex shader supported. Only vertex shaders with version numbers equal to or less than this value will be successfully created when calling the method **Direct3DDevice8.CreateVertexShader**. The level of vertex shader is specified to **CreateVertexShader** as the first token in the shader token stream.

The following table summarizes the supported vertex shader versions.

Version	**Functionality**
0.0	DirectX 7.0
1.0	DirectX 8.0 without address register A0.
1.1	DirectX 8.0 with address register A0.

The only difference between support levels 1.0 and 1.1 is support for the A0 register.

Setting the Vertex Shader Constant Registers

[C++]
You can use the following methods to set and retrieve the values in the vertex shader constant registers.

- **IDirect3DDevice8::GetVertexShaderConstant**
- **IDirect3DDevice8::SetVertexShaderConstant**

In addition, you can use the **def** Direct3DX assembler instruction to set the constant registers of a vertex shader.

[Visual Basic]
You can use the following methods to set and retrieve the values in the vertex shader constant registers.

- **Direct3DDevice8.GetVertexShaderConstant**
- **Direct3DDevice8.SetVertexShaderConstant**

In addition, you can use the **def** Direct3DX assembler instruction to set the constant registers of a vertex shader.

Compiling a Vertex Shader

[C++]
The Direct3DX utility library includes a set of functions to compile vertex shaders. The following functions are provided.

- **D3DXAssembleShader**
- **D3DXAssembleShaderFromFileA**
- **D3DXAssembleShaderFromFileW**

[Visual Basic]
The Direct3DX utility library provides a set of functions to compile pixel shaders. The following methods are provided.

- **D3DX8.AssembleShader**
- **D3DX8.AssembleShaderFromFile**

Creating a Vertex Shader

A vertex shader is defined by two token arrays that specify the declaration and function of the shader. The token arrays are composed of single or multiple **DWORD** tokens terminated by an 0xFFFFFFFF token value.

The shader declaration defines the static external interface of the shader, including binding of stream data to vertex register inputs and values loaded into the shader constant memory. The shader function defines the operation of the shader as an array of instructions. These instructions are executed in order for each vertex that is processed during the time that the shader is bound to a device. Shaders created without a function array apply the fixed-function vertex processing when that shader is current.

[C++]
The **IDirect3DDevice8::CreateVertexShader** method is used to create a vertex shader in Microsoft® DirectX® 8.0. You pass a declaration that defines the parallel DMA streams used and the function that defines the blending operations. Direct3D returns a shader handle. The shader validation is done when **CreateVertexShader** is called.

[Visual Basic]
The **Direct3DDevice8.CreateVertexShader** method is used to create a vertex shader in Microsoft® DirectX® 8.0. You pass a declaration that defines the parallel DMA streams used and the function that defines the blending operations. Microsoft® Direct3D® returns a shader handle. The shader validation is done when **CreateVertexShader** is called.

Vertex Shader Techniques and Applications

This section provides a list of common techniques implemented with vertex shaders. Information is divided into the following topics.

- Matrix Palette Skinning
- Procedural Geometry
- Wave Modeling

Matrix Palette Skinning

A set of eight to ten matrices can be loaded into the constant registers as a palette. Then each processed vertex can be transformed by a subset of these matrices and blended into a final result using a vertex-level weight. One convenient implementation is to use a packed 4-byte integer array to store the matrix palette addresses of four matrices influencing the vertex. A standard 4-D component, such as

a Microsoft® DirectX® 7.0 texture coordinate set, can contain the four weights used to blend those matrices.

Procedural Geometry

You can simulate a muscle moving under skin by an ellipsoid computed from a sphere with scaling along three axes. This can be evaluated and used to perturb the position of incoming mesh vertices. The muscle ellipsoid can be scaled and translated by updating the constant registers containing its relative position and scale factors.

Wave Modeling

A shape such as a quadratic hyperboloid can be generated in the unit square and mapped repeatedly onto incoming geometry. To do this, take input vertex (x, y) values, map them into unit square using the **frc** macro, compute local hyperboloid elevation, and add to input vertex z value. This can be used to simulate waves on an ocean, and so on.

Vertex Shader Examples

This section provides a list of common effects implemented with vertex shader code. Information is divided into the following topics.

- Transformation and Constant Shading
- Directional Light
- Other Vertex Shader Samples

Transformation and Constant Shading

The following shader performs a one-matrix transform with constant color. The input vertex register v0 is the position, which is assumed to be homogeneous. The transform matrix is in the constant registers (c[0] through c[3]). Diffuse color is loaded into c[4].

```
dp4 r0.x, v0, c[0]
dp4 r0.y, v0, c[1]
dp4 r0.z, v0, c[2]
dp4 r0.w, v0, c[3]
mov oD0, c[4]       ; The constant color.
mov oPos, r0        ; Emit the output.
```

This shader executes in six clock cycles.

Directional Light

The following shader builds on the Transformation and Constant Shading vertex shader example by adding a directional light source.

```
; The directional light source.
dp3 r1.x, v1, c[8]   ; N dot L
max r1, r1.x, c[0000]   ; clamp greater than 0

; diffuse = dot*diffuse + ambient
mad oD0, r1.x, c[5], c[6]
```

Other Vertex Shader Samples

Several samples in the SDK use vertex shader files. Look for files with the .vsh extension in the \Samples\Multimedia\Media directory.

X Files

This section discusses the structure of Microsoft® DirectX® (.x) files and how to use them in your applications.

Information is divided into the following topics.

- About X Files
- X File Interface Hierarchy
- X File Architecture
- Using X Files

About X Files

The .x file format was introduced with Microsoft® DirectX® 2.0. A binary version of this format was subsequently released with DirectX 3.0, which is also detailed in this reference. Then DirectX version 6.0 introduced interfaces and methods that enable reading from and writing to DirectX (.x) files.

The .x file format provides a rich, template-driven format that enables the storage of meshes, textures, animations, and user-definable objects. Support for animation sets enables you to store predefined paths for playback in real time. Instancing and hierarchies are also supported. Instancing enables multiple references to an object, such as a mesh, while storing its data only once per file. Hierarchies are used to express relationships between data records.

The .x file format provides low-level data primitives on which applications define higher-level primitives through templates.

X File Interface Hierarchy

[C++]
The following diagram illustrates the relationship between the .x file interfaces.

[Visual Basic]
The following diagram illustrates the relationship between the .x file classes.

X File Architecture

The Microsoft® DirectX® (.x) file format is architecture-free and context-free. It is template-driven and free of any usage knowledge. The file format can be used by any client application.

This following sections deal with the content and syntax of the file format. The file format uses the extension .x when used with the DirectX Software Development Kit (SDK).

- Reserved Words, Header, and Comments
- Templates
- Data
- Use of Commas and Semicolons

Reserved Words, Header, and Comments

The table below shows which words are reserved and must not be used.

ARRAY	DWORD	UCHAR
BINARY	FLOAT	ULONGLONG
BINARY_RESOURCE	SDWORD	UNICODE
CHAR	STRING	WORD
CSTRING	SWORD	
DOUBLE	TEMPLATE	

The variable-length header is compulsory and must be at the beginning of the data stream. The header contains the following data.

Type	Size	Contents	Content meaning
Magic Number (required)	4 bytes	"xof"	
Version Number (required)	2 bytes	03	Major Version 3

	2 bytes	02	Major Version 2
Format Type (required)	4 bytes	"txt"	Text File
		"bin"	Binary File
		"tzip"	MSZip Compressed Text File
		"bzip"	MSZip Compressed Binary File
Float Size (required)	4 bytes	0064	64-bit floats
		0032	32-bit floats

The following example shows a valid header.

```
xof 0302txt 0064
```

Comments are applicable only in text files. Comments can occur anywhere in the data stream. A comment begins with either C++ style double-slashes (//), or a number sign (#). The comment runs to the next new line. The following example shows valid comments.

```
#  This is a comment.
// This is another comment.
```

Templates

Templates define how the data stream is interpreted—the data is modulated by the template definition. This section discusses the following aspects of a template and provides an example template.

- Template Form, Name, and UUID
- Template Members
- Template Restrictions
- Template Example

There is one special template—the *header* template. It is recommended that each application define a header template and use it to define application-specific information, such as version information. If present, this header is read by the .x file format API. If a *flags* member is available, it is used to determine how the following data is interpreted. The *flags* member, if defined, should be a **DWORD**. One bit is currently defined—bit 0. If this bit is clear, the following data in the file is binary. If set, the following data is text. You can use multiple header data objects to switch between binary and text within the file.

Template Form, Name, and UUID

A template has the following form.

```
template <template-name> {
<UUID>
<member 1>;
```

```
...
<member n>;
[restrictions]
}
```

The template name is an alphanumeric name that can include the underscore character (_). It must not begin with a digit. The UUID is a universally unique identifier formatted to the Open Software Foundation's Distributed Computing Environment standard and enclosed by angle brackets (< >). For example: <3D82AB43-62DA-11cf-AB39-0020AF71E433>.

Template Members

Template members consist of a named data type followed by an optional name or an array of a named data type. Valid primitive data types are defined in the following table.

Type	Size
WORD	16 bits
DWORD	32 bits
FLOAT	IEEE float
DOUBLE	64 bits
CHAR	8 bits
UCHAR	8 bits
BYTE	8 bits
STRING	NULL terminated string
CSTRING	Formatted C string (not supported)
UNICODE	Unicode™ string (not supported)

Additional data types defined by templates encountered earlier in the data stream can also be referenced within a template definition. No forward references are allowed.

Any valid data type can be expressed as an array in the template definition. The basic syntax is shown in the following example.

```
array <data-type> <name>[<dimension-size>];
```

<dimension-size> can either be an integer or a named reference to another template member whose value is then substituted. Arrays can be n-dimensional, where n is determined by the number of paired square brackets trailing the statement, as in the following example.

```
array DWORD FixedHerd[24];
array DWORD Herd[nCows];
array FLOAT Matrix4x4[4][4];
```

Template Restrictions

Templates can be open, closed, or restricted. These restrictions determine which data types can appear in the immediate hierarchy of a data object defined by the template. An open template has no restrictions, a closed template rejects all data types, and a restricted template allows a named list of data types.

The syntax for indicating an open template is three periods enclosed by square brackets.

```
[ ... ]
```

A comma-separated list of named data types followed optionally by their universally unique identifiers (UUIDs) enclosed by square brackets indicates a restricted template.

```
[ { data-type [ UUID ] , } ... ]
```

The absence of either of the above indicates a closed template.

Template Example

The following shows an example template.

```
template Mesh {
<3D82AB44-62DA-11cf-AB39-0020AF71E433>
DWORD nVertices;
array Vector vertices[nVertices];
DWORD nFaces;
array MeshFace faces[nFaces];
 [ ... ]              // An open template
}
template Vector {
<3D82AB5E-62DA-11cf-AB39-0020AF71E433>
FLOAT x;
FLOAT y;
FLOAT z;
}                     // A closed template
template FileSystem {
<UUID>
STRING name;
[ Directory <UUID>, File <UUID> ]    // A restricted template
}
```

Data

Data objects contain the actual data or a reference to that data. Each data object has a corresponding template that specifies the data type. The following sections discuss the form and parts of data objects.

- Form, Identifier, and Name
- Data Members

Form, Identifier, and Name

Data objects have the following form.

```
<Identifier> [name] { [<UUID>]
<member 1>;
...
<member n>;
}
```

The identifier is compulsory and must match a previously defined data type or primitive. However, the name is optional. For details on the member part, see Data Members.

Data Members

Data members can be one of the following: data object, data reference, integer list, float list, or string list.

A data object is a nested data object. This enables the hierarchical nature of the file format to be expressed. The types of nested data objects allowed in the hierarchy may be restricted. See Templates for more information.

A data reference is a reference to a previously encountered data object as shown in the following example.

```
{
  name |
  UUID |
  name UUID
}
```

An integer list is a semicolon-separated list of integers, as shown in the following example.

```
1; 2; 3;
```

A float list is a semicolon-separated list of floats, as shown in the following example.

```
1.0; 2.0; 3.0;
```

A string list is a semicolon-separated list of strings, as shown in the following example.

```
"Moose"; "Goats"; "Sheep";
```

Use of Commas and Semicolons

Using commas and semicolons can be the most complex syntax issue in the file format, and this usage is very strict. Commas are used to separate array members; semicolons terminate each data item.

For example, if a template is defined in the following manner:

```
template mytemp {
DWORD myvar;
}
```

Then an instance of this template looks like the following:

```
mytemp dataTemp {
1;
}
```

If a template containing another template is defined in the following manner;

```
template mytemp {
DWORD myvar;
DWORD myvar2;
}
template container {
FLOAT aFloat;
mytemp aTemp;
}
```

Then an instance of this template looks like the following:

```
container dataContainer {
1.1;
2; 3;;
}
```

Note that the second line that represents the *mytemp* inside *container* has two semicolons at the end of the line. The first semicolon indicates the end of the data item, *aTemp* (inside container), and the second semicolon indicates the end of the *container*.

If an array is defined in the following manner:

```
Template mytemp {
array DWORD myvar[3];
}
```

Then an instance of this looks like the following:

```
mytemp aTemp {
```

```
1, 2, 3;
}
```

In the array example, there is no need for the data items to be separated by semicolons because they are delineated by commas. The semicolon at the end marks the end of the array.

Consider a template that contains an array of data items defined by a template.

```
template mytemp {
DWORD myvar;
DWORD myvar2;
}
template container {
DWORD count;
array mytemp tempArray[count];
}
```

An instance of this would look like the following example.

```
container aContainer {
3;
1;2;,3;4;,5;6;;
}
```

Using X Files

This section contains information about using .x files in a Microsoft® Direct3D® application. Information is divided into the following topics.

- Loading an X File
- Saving to an X File
- Creating a Cube
- Converting and Exporting 3-D Models to X Files

Loading an X File

[C++]
Use the following procedure to load an .x file.

1. Use the **DirectXFileCreate** function to create an **IDirectXFile** object.
2. If templates are present in the Microsoft® DirectX® file that you will load, use the **IDirectXFile::RegisterTemplates** method to register those templates.
3. Use the **IDirectXFile::CreateEnumObject** method to create an **IDirectXFileEnumObject** enumerator object.
4. Loop through the objects in the file. For each object, perform the following steps.

a. Use the **IDirectXFileEnumObject::GetNextDataObject** method to retrieve each **IDirectXFileData** object.
b. Use the **IDirectXFileData::GetType** method to retrieve the data's type.
c. Load the data using the **IDirectXFileData::GetData** method.
d. If the object has optional members, retrieve the optional members by calling the **IDirectXFileData::GetNextObject** method.
e. Release the **IDirectXFileData** object.

5. Release the **IDirectXFileEnumObject** object.
6. Release the **IDirectXFile** object.

[Visual Basic]
Use the following procedure to load an .x file.

1. Use the **DirectX8.DirectXFileCreate** method to create a **DirectXFile** object.
2. If templates are present in the Microsoft® DirectX® file that you will load, use the **DirectXFile.RegisterTemplates** method to register those templates.
3. Use the **DirectXFile.CreateEnumObject** method to create a **DirectXFileEnum** enumerator object.
4. Loop through the objects in the file. For each object, perform the following steps.
   a. Use the **DirectXFileEnum.GetNextDataObject** method to retrieve each **DirectXFileData** object.
   b. Use the **DirectXFileData.GetType** method to retrieve the data's type.
   c. Load the data using the **DirectXFileData.GetData** method.
   d. If the object has optional members, retrieve the optional members by calling the **DirectXFileData.GetNextObject** method.
   e. Set the **DirectXFileData** object variable to Nothing.
5. Set the **DirectXFileEnum** object variable to Nothing.
6. Set the **DirectXFile** object variable to Nothing.

Saving to an X File

[C++]
Use the following procedure to save .x file templates and data to an .x file.

1. Use the **DirectXFileCreate** function to create an **IDirectXFile** object.
2. Use the **IDirectXFile::RegisterTemplates** method to inform the Microsoft® DirectX® file system about any templates that you will use.
3. Use the **IDirectXFile::CreateSaveObject** method to create an **IDirectXFileSaveObject** object.

4. Use the **IDirectXFileSaveObject::SaveTemplates** method to save templates, if desired.
5. Loop through the objects to save. For each top-level object, perform the following steps.
   a. Use the **IDirectXFileSaveObject::CreateDataObject** method to create an **IDirectXFileData** object as a top-level object in the file. If the top-level data object has optional child objects, add them to the object by using the appropriate method from the next step.
   b. Each **IDirectXFileData** object can have optional child objects if its template allows it. The child objects can be any of the three types of objects: **IDirectXFileData**, **IDirectXFileDataReference**, or **IDirectXFileBinary**. Loop through the objects you need to save, adding each optional child member to the object list in the manner appropriate to its type, as illustrated in the following steps. Then, if the object type is Data, call the **IDirectXFileSaveObject::CreateDataObject** method to create an **IDirectXFileData** object, and then call the **IDirectXFileData::AddDataObject** method to add it as a child of the object. If the object type is Data Reference, call the **IDirectXFileData::AddDataReference** method to create and add the data reference object as a child of the object. Or, if the object type is Binary, call the **IDirectXFileData::AddBinaryObject** method to create and add the binary object as a child of the object.
   c. Call the **IDirectXFileSaveObject::SaveData** method to save the data object and its children.
   d. Release the **IDirectXFileData** object.
6. Release the **IDirectXFileSaveObject** object.
7. Release the **IDirectXFile** object.

[Visual Basic]
Use the following procedure to save .x file templates and data to an .x file.

1. Use the **DirectX8.DirectXFileCreate** method to create a **DirectXFile** object.
2. Use the **DirectXFile.RegisterTemplates** method to inform the Microsoft® DirectX® file system about any templates that you will use.
3. Use the **DirectXFile.CreateSaveObject** method to create an **DirectXFileSave** object.
4. Use the **DirectXFileSave.SaveTemplates** method to save templates, if desired.
5. Loop through the objects to save. For each top-level object, perform the following steps.
   a. Use the **DirectXFileSave.CreateDataObject** method to create a **DirectXFileData** object as a top-level object in the file. If the top-level data object has optional child objects, add them to the object by using the appropriate method as defined in the next step.

b. Each **DirectXFileData** object can have optional child objects if its template allows it. The child objects can be any of the three types of objects: **DirectXFileData**, **DirectXFileReference**, or **DirectXFileBinary**. Loop through the objects you need to save, adding each optional child member to the object list in the manner appropriate to its type, as illustrated in the following steps. Then, if the object type is Data, call the **DirectXFileSave.CreateDataObject** method to create an **DirectXFileData** object, and then call the **DirectXFileData.AddDataObject** method to add it as a child of the object. If the object type is Data Reference, call the **DirectXFileData.AddDataReference** method to create and add the data reference object as a child of the object. Or, if the object type is Binary, call the **DirectXFileData.AddBinaryObject** method to create and add the binary object as a child of the object.

c. Call the **DirectXFileSave.SaveData** method to save the data object and all its children.

d. Set the **DirectXFileData** object to Nothing.

6. Set the **DirectXFileSaveobject** to Nothing.
7. Set the **DirectXFileobject** to Nothing.

Creating a Cube

This section describes a simple cube and shows how to add textures, frames, and animations.

- Defining a Simple Cube
- Adding Textures
- Adding Frames and Animations

Defining a Simple Cube

The following file defines a simple cube that has four red sides and two green sides. In this file, optional information is used to add information to the data object defined by the **Mesh** template.

```
Material RedMaterial {
1.000000;0.000000;0.000000;1.000000;;    // R = 1.0, G = 0.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
}
Material GreenMaterial {
0.000000;1.000000;0.000000;1.000000;;    // R = 0.0, G = 1.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
```

```
0.000000;0.000000;0.000000;;
}
// Define a mesh with 8 vertices and 12 faces (triangles). Use
// optional data objects in the mesh to specify materials, normals,
// and texture coordinates.
Mesh CubeMesh {
8;                          // 8 vertices.
1.000000;1.000000;-1.000000;,     // Vertex 0.
-1.000000;1.000000;-1.000000;,    // Vertex 1.
-1.000000;1.000000;1.000000;,     // And so on.
1.000000;1.000000;1.000000;,
1.000000;-1.000000;-1.000000;,
-1.000000;-1.000000;-1.000000;,
-1.000000;-1.000000;1.000000;,
1.000000;-1.000000;1.000000;;

12;                  // 12 faces.
3;0,1,2;,              // Face 0 has three vertices.
3;0,2,3;,              // And so on.
3;0,4,5;,
3;0,5,1;,
3;1,5,6;,
3;1,6,2;,
3;2,6,7;,
3;2,7,3;,
3;3,7,4;,
3;3,4,0;,
3;4,7,6;,
3;4,6,5;;

// All required data has been defined. Now define optional data
// using the hierarchical nature of the file format.
MeshMaterialList {
2;                  // Number of materials used.
12;                  // A material for each face.
0,                  // Face 0 uses the first material.
0,
0,
0,
0,
0,
0,
1,                  // Face 8 uses the second material.
1,
1,
```

```
1;;
{RedMaterial}        // References to the definitions
{GreenMaterial}      // of material 0 and 1.
}
MeshNormals {
8;                   // Define 8 normals.
0.333333;0.666667;-0.666667;,
-0.816497;0.408248;-0.408248;,
-0.333333;0.666667;0.666667;,
0.816497;0.408248;0.408248;,
0.666667;-0.666667;-0.333333;,
-0.408248;-0.408248;-0.816497;,
-0.666667;-0.666667;0.333333;,
0.408248;-0.408248;0.816497;;
12;                  // For the 12 faces, define the normals.
3;0,1,2;,
3;0,2,3;,
3;0,4,5;,
3;0,5,1;,
3;1,5,6;,
3;1,6,2;,
3;2,6,7;,
3;2,7,3;,
3;3,7,4;,
3;3,4,0;,
3;4,7,6;,
3;4,6,5;;
}
MeshTextureCoords {
8;                   // Define texture coords for each vertex.
0.000000;1.000000;
1.000000;1.000000;
0.000000;1.000000;
1.000000;1.000000;
0.000000;0.000000;
1.000000;0.000000;
0.000000;0.000000;
1.000000;0.000000;;
}
}
```

Adding Textures

To add textures, use the hierarchical nature of the file format and add an optional **TextureFilename** data object to the **Material** data objects. The **Material** objects now read as follows:

```
Material RedMaterial {
1.000000;0.000000;0.000000;1.000000;;    // R = 1.0, G = 0.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
TextureFilename {
"tex1.ppm";
}
}
Material GreenMaterial {
0.000000;1.000000;0.000000;1.000000;;     // R = 0.0, G = 1.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
TextureFilename {
"win95.ppm";
}
}
```

Adding Frames and Animations

This section shows how to add frames and animations to a simple cube.

- Working with Frames
- Working with AnimationSets and Animations

Working with Frames

A frame is expected to take the following structure.

```
Frame Aframe {       // The frame name is chosen for convenience.
FrameTransformMatrix {
...transform data...
}
[ Meshes ] and/or [ More frames]
}
```

Place the defined cube mesh inside a frame with an identity transform. Then apply an animation to this frame.

```
Frame CubeFrame {
FrameTransformMatrix {
1.000000, 0.000000, 0.000000, 0.000000,
0.000000, 1.000000, 0.000000, 0.000000,
0.000000, 0.000000, 1.000000, 0.000000,
0.000000, 0.000000, 0.000000, 1.000000;;
}
{CubeMesh}       // You could have the mesh inline, but this
```

```
            // uses an object reference instead.
}
```

Working with AnimationSets and Animations

An animation is defined by a set of keys. A key is a time value associated with a scaling operation, an orientation, or a position.

```
Animation Animation0 {       // The name is chosen for convenience.
{ Frame that it applies to&em;normally a reference }
AnimationKey {
...animation key data...
}
{ ...more animation keys... }
}
```

Animations are then grouped into AnimationSets:

```
AnimationSet AnimationSet0 { // The name is chosen for convenience.
{ an animation—could be inline or a reference }
{ ... more animations ... }
}
```

Now take the cube through an animation.

```
AnimationSet AnimationSet0 {
Animation Animation0 {
{CubeFrame}    // Use the frame containing the cube.
AnimationKey {
2;             // Position keys
9;             // 9 keys
10; 3; -100.000000, 0.000000, 0.000000;;,
20; 3; -75.000000, 0.000000, 0.000000;;,
30; 3; -50.000000, 0.000000, 0.000000;;,
40; 3; -25.500000, 0.000000, 0.000000;;,
50; 3; 0.000000, 0.000000, 0.000000;;,
60; 3; 25.500000, 0.000000, 0.000000;;,
70; 3; 50.000000, 0.000000, 0.000000;;,
80; 3; 75.500000, 0.000000, 0.000000;;,
90; 3; 100.000000, 0.000000, 0.000000;;;
}
}
}
```

For more information, see the **Animation** and **AnimationSet** templates.

Converting and Exporting 3-D Models to X Files

Microsoft® Direct3D® supplies the following tools to convert and export 3-D models to the .x file format.

- Conv3ds.exe
- XSkinExp.dle

Conv3ds.exe

The Conv3ds.exe utility converts three-dimensional (3-D) models produced by Autodesk 3-D Studio and other modeling packages to the Microsoft® DirectX® file format. By default, the utility produces binary .x files with no templates.

Information is divided into the following sections.

- Running Conv3ds.exe
- Specifying Optional Arguments for Conv3ds.exe
- Producing 3DS Files from Lightwave Objects
- Hints and Tips

Running Conv3ds.exe

You can run Conv3ds.exe with no options, and it will produce an .x file containing a hierarchy of frames. For example, the following command produces an .x file called File.x from the .3ds file.

```
conv3ds File.3ds
```

To run Conv3ds.exe with options, see Specifying Optional Arguments for Conv3ds.exe.

Specifying Optional Arguments for Conv3ds.exe

If the .3ds file contains key frame data, you can use the **-A** option to produce an .x file that contains an animation set. The following command would do this.

```
conv3ds -A File.3ds
```

The *File.3ds* parameter is the name of the file to be converted.

Use the **-m** option to make an .x file that contains a single mesh made from all the objects in the .3ds file.

```
conv3ds -m File.3ds
```

Use the **-T** option to wrap all the objects and frame hierarchies in a single top-level frame. Using this option, all the frames and objects in the .3ds file can be loaded with a single call. The first top-level frame hierarchy in the .x file will be loaded. The frame containing all the other frames and meshes is called x3ds_filename, without the .3ds extension. The **-T** option will have no effect if it is used with the **-m** option.

The **-s** option enables you to specify a scale factor for all the objects converted in the .3ds file. For example, the following command makes all objects ten times bigger.

```
conv3ds -s10 File.3ds
```

The following command makes all objects ten times smaller:

```
conv3ds -s0.1 File.3ds
```

The **-r** option reverses the winding order of the faces when the .3ds file is converted. If, after converting the .3ds file and viewing it in Microsoft® Direct3D®, the object appears inside-out, try converting it with the **-r** option. All Lightwave models exported as .3ds files need this option. For details, see Producing 3DS Files from Lightwave Objects.

The **-v** option turns on verbose output mode. Specify an integer with it. The following table shows the currently supported integers.

Option	**Meaning**
-v0	Default. Verbose mode off.
-v1	Prints warnings about bad objects, and prints general information about what the co doing.
-v2	Prints basic key frame information, the objects being included in the conversion pro information about the objects being saved.
-v3	Very verbose. Most useful for debugging information.

The **-e** option enables you to change the extension for texture map files, as shown in the following example.

```
conv3ds -e"ppm" File.3ds
```

If File.3ds contains objects that reference the texture map file Brick.gif, the .x file will reference the texture map file Brick.ppm. The converter does not convert the texture map file. The texture map files must be in the D3DPATH when the resulting .x file is loaded. The D3DPath is an environment variable that sets the default search path.

The **-x** option forces the Conv3ds.exe utility to produce a text .x file, instead of a binary .x file. Text files are larger but can be easily edited by hand.

The **-X** option forces the Conv3ds.exe utility to include the .x file templates in the file. By default, the templates are not included.

The **-t** option specifies that the .x file produced will not contain texture information.

The **-N** option specifies that the .x file produced will not contain normal vector information. All the load calls will generate normal vectors for objects with no normal vectors in the .x file.

The **-c** option specifies that the .x file produced should not contain texture coordinates. By default, if you use the **-m** option, the mesh that is output will contain (0,0) uv texture coordinates if the .3ds object had no texture coordinates.

The **-f** option specifies that the .x file produced should not contain a frame transformation matrix.

The **-z** and **-Z** options enable you to adjust the alpha face color value of all the materials referenced by objects in the .x file. For example, the following command causes Conv3ds.exe to add 0.1 to all alpha values under 0.2.

```
conv3ds -z0.1 -Z0.2 File.3ds
```

The following command causes Conv3ds.exe to subtract 0.2 from the alpha values for all alphas.

```
conv3ds-z"-0.2" -z1 File.3ds
```

The **-o** option enables you to specify the file name for the .x file produced.

The **-h** option tells the converter not to try to resolve any hierarchy information in the .3ds file, usually produced by the key framer. Instead, all the objects are output in top-level frames if the **-m** option is not used.

Producing 3DS Files from Lightwave Objects

There are several issues with .3ds files exported by the Trans3d plug-in for Lightwave. These are best handled using the following Conv3ds.exe command.

```
conv3ds -r -N -f -h -T|m trans3dfile.3ds
```

All the .3ds objects produced by Trans3d and the Lightwave object editor need their winding order reversed. Otherwise, they appear inside-out when displayed. They contain no surface normal vector information.

Hints and Tips

If you cannot see objects produced by Conv3ds.exe after loading them, use the scale option **-s** with a scale factor of approximately 100. This increases the scale of the objects in the .x file.

If, after loading the object into the viewer and switching from flat shading into Gouraud shading, the object turns dark gray, try converting with the **-N** option.

If the textures are not loaded after the object is converted, check whether the object is referencing either .ppm or .bmp files by using the **-e** option. Also check whether the texture widths and heights are a power of 2. Make sure the textures are stored in a directory in your D3DPATH.

Currently, Conv3ds.exe cannot handle dummy frames used in .3ds animations. It ignores them. However, it converts any child objects.

XSkinExp.dle

XSkinExp.dle is located in the extras directory and is used as an exporter for 3dsmax. To find out how to use it, read the readme file provided with XSkinExp.dle.

Programming Tips and Tools

This section contains information to help you develop Microsoft® DirectX® Graphics applications efficiently. The following topics are covered.

- Troubleshooting
- Performance Optimizations
- Multithreading Issues
- Working with Device Windows
- Working with Earlier Drivers
- Working with Multiple Monitor Systems
- Presenting Multiple Views in Windowed Mode

Troubleshooting

This section lists common categories of problems that you might encounter when writing Microsoft® Direct3D® applications, and how to prevent them.

- Device Creation
- Using Lit Vertices
- Nothing Visible
- Debugging
- Borland Floating-Point Initialization
- Parameter Validation

Device Creation

[C++]
If your application fails during device creation, check for the following common errors.

- Make sure you check the device capabilities, particularly the render depths.
- Examine the error code. **D3DERR_OUTOFVIDEOMEMORY** is always possible.
- Use the debug DirectX DLLs and review output messages under the debugger.

[Visual Basic]
If your application fails during device creation, check for the following common errors.

- Make sure you check the device capabilities, particularly the render depths.
- Examine the error code. **D3DERR_OUTOFVIDEOMEMORY** is always possible.
- Use the debug DirectX DLLs and review output messages under the debugger.

Using Lit Vertices

[C++]
Applications that use lit vertices should disable the Microsoft® Direct3D® lighting engine by setting the D3DRS_LIGHTING render state to FALSE. By default, when lighting is enabled, the system sets the color for any vertex that doesn't contain a normal vector to 0 (black), even if the input vertex contained a nonzero color value. Because lit-vertices don't, by their nature, contain a vertex normal, any color information passed to Direct3D is lost during rendering if the lighting engine is enabled.

Obviously, vertex color is important to any application that performs its own lighting. To prevent the system from imposing the default values, make sure you set D3DRS_LIGHTING to FALSE.

[Visual Basic]
Applications that use lit vertices, analogous to the **D3DLVERTEX** type, should disable the Microsoft® Direct3D® lighting engine by setting the D3DRS_LIGHTING render state to False. By default, when lighting is enabled, the system sets the color for any vertex that doesn't contain a normal vector to 0 (black), even if the input vertex contained a nonzero color value. Because lit-vertices such as **D3DLVERTEX** don't, by their nature, contain a vertex normal, any color information passed to Direct3D is lost during rendering if the lighting engine is enabled.

Obviously, vertex color is important to any application that performs its own lighting. To prevent the system from imposing the default values, make sure you set D3DRS_LIGHTING to False.

Nothing Visible

If your application runs but nothing is visible, check for the following common errors.

- Ensure that your triangles are not degenerate.
- Ensure that your triangles are not being culled.
- Make sure that your transformations are internally consistent.
- Check the viewport settings to be sure they allow your triangles to be seen.

Debugging

[Visual Basic]

Note
Information in this topic pertains only to applications developed in C/C++.

[C++]
Debugging a Microsoft® Direct3D® application can be challenging. Try the following techniques, in addition to checking all the return values—a particularly important piece of advice in Direct3D programming, which is dependent on very different hardware implementations.

- Switch to debug DLLs.
- Force a software-only device, turning off hardware acceleration even when it is available.
- Force surfaces into system memory.
- Create an option to run in a window, so that you can use an integrated debugger.

The second and third options in this list can help you avoid the Win16 lock which can otherwise cause your debugger to hang.

Also, try adding the following entries to Win.ini.

```
[Direct3D]
debug=3
[DirectDraw]
debug=3
```

Borland Floating-Point Initialization

[Visual Basic]

Note
Information in this topic pertains only to applications developed in C/C++.

[C++]
Compilers from Borland report floating-point exceptions in a manner that is incompatible with Microsoft® Direct3D®. To solve this problem, include a _matherr exception handler like the following:

```
// Borland floating point initialization
#include <math.h>
#include <float.h>

void initfp(void)
{
   // Disable floating point exceptions.
   _control87(MCW_EM,MCW_EM);
}

int _matherr(struct _exception  *e)
{
   e;             // Dummy reference to catch the warning.
   return 1;      // Error has been handled.
}
```

Parameter Validation

[Visual Basic]

Note
Information in this topic pertains only to applications developed in C/C++.

[C++]
For performance reasons, the debug version of the Microsoft® Direct3D® Immediate Mode run time performs more parameter validation than the retail version, which sometimes performs no validation at all. This enables applications to perform robust debugging with the slower debug run-time component before using the faster retail version for performance tuning and final release.

Although several Direct3D Immediate Mode methods impose limits on the values that they can accept, these limits are often only checked and enforced by the debug version of the Direct3D Immediate Mode run time. Applications must conform to these limits, or unpredictable and undesirable results can occur when running on the retail version of Direct3D. For example, the **IDirect3Ddevice8::DrawPrimitive** method accepts a parameter (*PrimitiveCount*) that indicates the number of primitives that the method will render. The method can only accept values between 0 and D3DMAXNUMPRIMITIVES. In the debug version of Direct3D, if you pass more than D3DMAXNUMPRIMITIVES primitives, the method fails gracefully, printing

an error message to the error log, and returning an error value to your application. Conversely, if your application makes the same error when it is running with the retail version of the run time, behavior is undefined. For performance reasons, the method does not validate the parameters, resulting in unpredictable and completely situational behavior when they are not valid. In some cases the call might work, and in other cases it might cause a memory fault in Direct3D. If an invalid call consistently works with a particular hardware configuration and DirectX version, there is no guarantee that it will continue to function on other hardware or with future releases of DirectX.

If your application encounters unexplained failures when running with the retail Direct3D Immediate Mode run-time file, test against the debug version and look closely for cases where your application passes invalid parameters.

Performance Optimizations

Every developer who creates real-time applications that use 3-D graphics is concerned about performance optimization. This section provides you with guidelines about getting the best performance from your code.

You can use the guidelines in the following sections for any Microsoft® Direct3D® application.

- General Performance Tips
- Databases and Culling
- Batching Primitives
- Lighting Tips
- Texture Size
- Using Dynamic Vertex and Index Buffers
- Using Meshes
- Z-buffer Performance

General Performance Tips

You can follow a few general guidelines to increase the performance of your application.

- Only clear when you must.
- Minimize state changes.
- Use perspective correction only if you must.
- Use smaller textures, if you can do so.
- Gracefully degrade special effects that require a disproportionate share of system resources.

- Constantly test your application's performance.
- Ensure that your application runs well with both hardware acceleration and software emulation.

Databases and Culling

Building a reliable database of the objects in your world is key to excellent performance in Microsoft® Direct3D®. It is more important than improvements to rasterization or hardware.

You should maintain the lowest polygon count you can possibly manage. Design for a low polygon count by building low-poly models from the start. Add polygons if you can do so without sacrificing performance later in the development process. Remember, the fastest polygons are the ones you don't draw.

Batching Primitives

To get the best rendering performance during execution, try to work with primitives in batches and keep the number of render-state changes as low as possible. For example, if you have an object with two textures, group the triangles that use the first texture and follow them with the necessary render state to change the texture. Then group all the triangles that use the second texture. The simplest hardware support for Microsoft® Direct3D® is called with batches of render states and batches of primitives through the hardware abstraction layer (HAL). The more effectively the instructions are batched, the fewer HAL calls are performed during execution.

Lighting Tips

[C++]
Because lights add a per-vertex cost to each rendered frame, you can achieve significant performance improvements by being careful about how you use them in your application. Most of the following tips derive from the maxim, the fastest code is code that is never called.

- Use as few light sources as possible. If you only need to increase the overall lighting level, use the ambient light instead of adding a new light source. It's much cheaper.
- Directional lights are cheaper than point lights or spotlights. For directional lights, the direction to the light is fixed and doesn't need to be calculated on a per-vertex basis.
- Spotlights can be cheaper than point lights, because the area outside the cone of light is calculated quickly. Whether spotlights are cheaper depends on how much of your scene is lit by the spotlight.
- Use the range parameter to limit your lights to only the parts of the scene you need to illuminate. All the light types exit fairly early when they are out of range.

- Specular highlights almost double the cost of a light—use them only when you must. Set the D3DRS_SPECULARENABLE render state to 0, the default value, whenever possible. When defining materials you must set the specular power value to zero to turn off specular highlights for that material—simply setting the specular color to 0,0,0 is not enough.

[Visual Basic]
Because lights add a per-vertex cost to each rendered frame, you can achieve significant performance improvements by being careful about how you use them in your application. Most of the following tips derive from the maxim, the fastest code is code that is never called.

- Use as few light sources as possible. If you only need to bring up the overall level of lighting, use the ambient light instead of adding a new light source. It's much cheaper.
- Directional lights are cheaper than point lights or spotlights. For directional lights, the direction to the light is fixed and doesn't need to be calculated on a per-vertex basis.
- Spotlights can be cheaper than point lights, because the area outside of the cone of light is calculated quickly. Whether spotlights are cheaper depends on how much of your scene is lit by the spotlight.
- Use the range parameter to limit your lights to only the parts of the scene you need to illuminate. All the light types exit fairly early when they are out of range.
- Specular highlights almost double the cost of a light—use them only when you must. Set the D3DRS_SPECULARENABLE render state to 0, the default value, whenever possible. When defining materials you must set the specular power value to zero to turn off specular highlights for that material—simply setting the specular color to 0,0,0 is not enough.

Texture Size

Texture-mapping performance is heavily dependent on the speed of memory. There are a number of ways to maximize the cache performance of your application's textures.

- Keep the textures small; the smaller the textures are, the better chance they have of being maintained in the main CPU's secondary cache.
- Do not change the textures on a per-primitive basis. Try to keep polygons grouped in order of the textures they use.
- Use square textures whenever possible. Textures whose dimensions are 256×256 are the fastest. If your application uses four 128×128 textures, for example, try to ensure that they use the same palette and place them all into one 256×256

texture. This technique also reduces the amount of texture swapping. Of course, you should not use 256×256 textures unless your application requires that much texturing because, as mentioned, textures should be kept as small as possible.

Using Dynamic Vertex and Index Buffers

Dynamic vertex and index buffers have a difference in performance based the size and usage. The usage styles below help to determine whether to use D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

Usage Style 1:

```
for loop()
{
   pBuffer->Lock(...D3DLOCK_DISCARD...); //Ensures that hardware
                                         //doesn't stall by returning
                                         //a new pointer.
   Fill data (optimally 1000s of vertices/indices, no fewer) in pBuffer.
   pBuffer->Unlock()
   Change state(s).
   DrawPrimitive() or DrawIndexedPrimitive()
}
```

Usage Style 2:

```
for loop()
{
   pVB->Lock(...D3DLOCK_DISCARD...); //Ensures that hardware doesn't
                                     //stall by returning a new
                                     //pointer.
   Fill data (optimally 1000s of vertices/indices, no fewer) in pBuffer.
   for loop( 100s of times )
   {
      Change State
      DrawPrimitive() or DrawIndexPrimitives() //Tens of primitives
   }
}
```

Usage Style 3:

```
for loop()
{
   If there is space in the Buffer
   {
      //Append vertices/indices.
      pBuffer->Lock(…D3DLOCK_NOOVERWRITE…);
   }
```

```
    Else
    {
       //Reset to beginning.
       pBuffer->Lock(…D3DLOCK_DISCARD…);
    }
    Fill few 10s of vertices/indices in pBuffer
    pBuffer->Unlock
    Change State
    DrawPrimitive() or DrawIndexedPrimitive() //A few primitives
}
```

Style 1 is faster than either style 2 or 3, but is generally not very practical. Style 2 is usually faster than style 3, provided that the application fills at least a couple thousand vertices/indices for every **Lock,** on average. If the application fills fewer than that on average, then style 3 is faster. There is no guaranteed answer as to which lock method is faster and the best way to find out is to experiment.

Using Meshes

You can optimize meshes by using Microsoft® Direct3D® indexed triangles instead of indexed triangle strips. The hardware will discover that 95% of successive triangles actually form strips and adjust accordingly. Many drivers do this for legacy hardware also.

The following topics describes the different attributes of a mesh.

Attribute ID

An attribute ID is a value that associates a group of faces with an attribute group. This ID describes which subset of faces **DrawSubset** should draw. Attribute IDs are specified for the faces in the attribute buffer. The actual values of the attribute IDs can be anything that fits in 32bits, but it is common to use 0 to n where n is the number of attributes.

Attribute Buffer

The attribute buffer is an array of **DWORD**s (one per face) that specifies which attribute group each face belongs in. This buffer is initialized to zero on creation of a mesh, but is either filled by the load routines or must be filled by the user if more than one attribute with ID 0 is desired. This buffer contains the information that is used to sort the mesh based on attributes in **Optimize**. If no attribute table is present, **DrawSubset** scans this buffer to select the faces of the given attribute to draw.

Attribute Table

The attribute table is a structure owned and maintained by the mesh. The only way for one to be generated is by calling **Optimize** with attribute sorting or stronger optimization enabled. The attribute table is used to quickly initiate a single draw primitive call to **DrawSubset**. The only other use is that progressing meshes also maintain this structure, so it is possible to see what faces and vertices are active at the current level of detail(LOD).

Z-Buffer Performance

Applications can increase performance when using z-buffering and texturing by ensuring that scenes are rendered from front to back. Textured z-buffered primitives are pretested against the z-buffer on a scan line basis. If a scan line is hidden by a previously rendered polygon, the system rejects it quickly and efficiently. Z-buffering can improve performance, but the technique is most useful when a scene includes a great deal of *overdraw*. Overdraw is the average number of times that a screen pixel is written to. Overdraw is difficult to calculate exactly, but you can often make a close approximation. If the overdraw averages less than 2, you can achieve the best performance by turning z-buffering off and rendering the scene from back-to-front.

On faster personal computers, software rendering to system memory is often faster than rendering to video memory, although it has the disadvantage of not being able to use double buffering or hardware-accelerated clear operations. If your application can render to either system or video memory, and if you include a routine that tests which is faster, you can take advantage of the best approach on the current system. The Microsoft® Direct3D® sample code in this SDK demonstrates this strategy. It is necessary to implement both methods because there is no other way to test the speed. Speeds can vary enormously from computer to computer, depending on the main-memory architecture and the type of graphics adapter being used.

Multithreading Issues

[Visual Basic]
This topic pertains only to application development in C++.

[C++]
Full-screen Microsoft® Direct3D® applications provide a window handle to the Direct3D run time. The window is hooked by the run time. This means that all messages passed to the application's window message procedure have first been examined by the Direct3D run time's own message-handling procedure.

Display mode changes are affected by support routines built into the underlying operating system. When mode changes occur, the system broadcasts several messages to all applications. In Direct3D applications, the messages are received on the window procedure thread, which is not necessarily the same thread that called **IDirect3DDevice8::Reset** or **IDirect3D8::CreateDevice** (or the final **IUnknown::Release** of **IDirect3DDevice8**, which can cause a display mode change). The Direct3D run time maintains several critical sections internally. Because at least one of these critical sections is held across the mode switch caused by **Reset** or **CreateDevice**, these critical sections are still held when the application receives the mode-change related window messages.

This design has some implications for multithreaded applications. In particular, an application must be sure to strongly segregate its window message handling threads

from its Direct3D threads. An application that causes a mode change on one thread but makes Direct3D calls on a different thread in its window procedure is in danger of deadlock.

For these reasons, Direct3D is designed so that the methods **Reset**, **CreateDevice**, **TestCoorperativeLevel**, or the final **Release** of **IDirect3DDevice8** can only be called from the same thread that handles window messages.

Working with Device Windows

[Visual Basic]
This topic pertains only to application development in C++.

[C++]
This section lists issues that you might encounter when working with device windows in Microsoft® Direct3D® applications.

- Direct3D only hooks up the focus windows instead of the device window with the Direct3D message processing function, and only processes the focus window messages. So, the focus window should be the parent of any device window.
- For any application, multiple monitor or single monitor, at least one device window must be the focus window. In practical terms, this means that you cannot use a child window of your focus window as the device window in single monitor systems. You must use the parent focus window as both the focus and device windows. For a multiple monitor system, at least one monitor must use the focus window as its device window.

Working with Earlier Drivers

This section lists issues that may be encountered when working with Microsoft® DirectX® 8.0 on drivers written for versions of DirectX earlier than DirectX 8.0.

- When working with a TnLHAL device, the fog intensity is computed but the absolute value operation is not applied to this value. Rather, the value is left negative if that is what is computed. The best way to avoid this situation is to set up transforms appropriately. A less-preferred method is to make the fog-start and fog-end values negative to match.

[C++]
You can detect a driver not written for DirectX 8.0 by querying the **MaxStream** member of **D3DCAPS8**. If this value is 0, then it is not a DirectX 8.0 driver.

[Visual Basic]
You can detect a driver not written for DirectX 8.0 by querying the **MaxStream** member of **D3DCAPS8**. If this value is 0, then it is not a DirectX 8.0 driver.

Working with Multiple Monitor Systems

[C++]
The concept of exclusive full-screen mode is retained in Microsoft® DirectX® 8.0, but it is kept entirely implicit in the **IDirect3D8::CreateDevice** and **IDirect3DDevice8::Reset** method calls. Whenever a device is successfully reset or created in full-screen operation, the Microsoft® Direct3D® object that created the device is marked as owning all adapters on that system. This state is known as exclusive mode, and at this point the Direct3D object owns exclusive mode. Exclusive mode means that devices created by any other Direct3D8 object can neither assume full-screen operation nor allocate video memory. In addition, when a Direct3D8 object assumes exclusive mode, all devices other than the device that went full-screen are placed into the lost state. For information on how to handle lost devices, see Lost Devices.

[Visual Basic]
The concept of exclusive full-screen mode is retained in Microsoft® DirectX® 8.0, but it is kept entirely implicit in the **Direct3D8.CreateDevice** and **Direct3DDevice8.Reset** method calls. Whenever a device is successfully reset or created in full-screen operation, the Direct3D8 object that created the device is marked as owning all adapters on that system. This state is known as exclusive mode, and at this point the Direct3D object owns exclusive mode. Exclusive mode means that devices created by any other Direct3D8 object can neither assume full-screen operation nor allocate video memory. In addition, when a Direct3D8 object assumes exclusive mode, all devices other than the device that went full-screen are placed into the lost state. For information on how to handle lost devices, see Lost Devices.

Along with exclusive mode, the Direct3D8 object is informed of the focus window to be used by the device. Exclusive mode is released when the last full-screen device owned by that Direct3D8 object is either reset to windowed mode or destroyed.

Devices can be divided into two categories when a Direct3D8 object owns exclusive mode. The first category of devices are those that were created by the same Direct3D8 object that created the device that is already full-screen, have the same focus window as the device that is already full-screen, and represent a different adapter from any full-screen device. Devices in this category have no restrictions

concerning their ability to be reset or created and are not placed into the lost state. Devices in this category can even be placed into full-screen mode.

Devices not in this category, which would be those created by a different Direct3D8 object, or with a different focus window, or for some adapter with a device already full-screen cannot be reset and remain in the lost state until exclusive mode is lost.

The practical implication is that a multiple monitor application can place several devices in full-screen mode, but only if all these devices are for different adapters, were created by the same Direct3D8 object, and all share the same focus window.

Presenting Multiple Views in Windowed Mode

[C++]
In addition to the swap chain that is owned and manipulated through the **IDirect3DDevice8** interface, an application can create additional swap chains in order to present multiple views from the same device. The application typically creates one swap chain per view by using the **IDirect3DDevice8::CreateAdditionalSwapChain** method, and associates each swap chain with a particular window. The application renders images into the back buffers of each swap chain, and then presents them individually.

Only one swap chain at a time can be full-screen on each adapter.

[Visual Basic]
In addition to the swap chain owned and manipulated through the **Direct3DDevice8** interface, an application can create additional swap chains in order to present multiple views from the same device. The application typically creates one swap chain per view by using the **Direct3DDevice8.CreateAdditionalSwapChain** method, and associates each swap chain with a particular window. The application renders images into the back buffers of each swap chain, and then presents them individually.

Only one swap chain at a time can be full-screen on each adapter.

DirectX Graphics C/C++ Tutorials

The tutorials in this section show how to use Microsoft® Direct3D® and Direct3DX in a C/C++ application for common tasks. The tasks are divided into required steps. In some cases, steps are organized into substeps for clarity.

The following tutorials are provided.

- Tutorial 1: Creating a Device
- Tutorial 2: Rendering Vertices
- Tutorial 3: Using Matrices
- Tutorial 4: Creating and Using Lights
- Tutorial 5: Using Texture Maps
- Tutorial 6: Using Meshes

Note

The sample code in these tutorials is from source projects whose location is provided in each tutorial.
The sample files in these tutorials are written in C++. If you are using a C compiler, you must make the appropriate changes to the files for them to successfully compile. At the very least, you need to add the *vtable* and *this* pointers to the interface methods.
Some comments in the included sample code might differ from the source files in the Microsoft Platform Software Development Kit (SDK). Changes are made for brevity only and are limited to comments to avoid changing the behavior of the sample code.

See Also

DirectX Graphics C/C++ Samples

Tutorial 1: Creating a Device

To use Microsoft® Direct3D®, you first create an application window, then you create and initialize Direct3D objects. You use the COM interfaces that these objects implement to manipulate them and to create other objects required to render a scene. The CreateDevice sample project on which this tutorial is based illustrates these tasks by creating a Direct3D device and rendering a blue screen.

This tutorial uses the following steps to initialize Direct3D, render a scene, and eventually shut down.

- Step 1: Creating a Window
- Step 2: Initializing Direct3D
- Step 3: Handling System Messages
- Step 4: Rendering and Displaying a Scene
- Step 5: Shutting Down

Note

The path of the CreateDevice sample project is:
(SDK Root)\Samples\Multimedia\Direct3D\Tutorials\Tut01_CreateDevice.

Step 1: Creating a Window

The first thing any Microsoft® Windows® application must do when it is executed is create an application window to display to the user. To do this, the CreateDevice sample project begins execution at its **WinMain** function. The following sample code performs window initialization.

```
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )
{
    // Register the window class.
    WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                      GetModuleHandle(NULL), NULL, NULL, NULL, NULL,
                      "D3D Tutorial", NULL };
    RegisterClassEx( &wc );

    // Create the application's window.
    HWND hWnd = CreateWindow( "D3D Tutorial", "D3D Tutorial 01: CreateDevice",
                              WS_OVERLAPPEDWINDOW, 100, 100, 300, 300,
                              GetDesktopWindow(), NULL, wc.hInstance, NULL );
```

The preceding code sample is standard Windows programming. The sample starts by defining and registering a window class called "D3D Tutorial." After the class is registered, the sample code creates a basic top-level window that uses the registered class, with a client area of 300 pixels wide by 300 pixels tall. This window has no menu or child windows. The sample uses the WS_OVERLAPPEDWINDOW window style to create a window that includes Minimize, Maximize, and Close buttons common to windowed applications. (If the sample were to run in full-screen mode, the preferred window style is WS_EX_TOPMOST, which specifies that the created window should be placed above all non-topmost windows and should stay above them, even when the window is deactivated). Once the window is created, the code sample calls standard Microsoft Win32® functions to display and update the window.

With the application window ready, you can begin setting up the essential Microsoft Direct3D® objects, as described in Step 2: Initializing Direct3D.

Step 2: Initializing Direct3D

The CreateDevice sample project performs Microsoft® Direct3D® initialization in the **InitD3D** application-defined function called from **WinMain** after the window is created. After you create an application window, you are ready to initialize the Direct3D object that you will use to render the scene. This process includes creating a Direct3D object, setting the presentation parameters, and finally creating the Direct3D device.

After creating a Direct3D object, you can use the **IDirect3D8::CreateDevice** method to create a Direct3D device. You can also use the Direct3D object to enumerate

devices, types, modes, and so on. The code fragment below creates a Direct3D object with the **Direct3DCreate8** function.

```
if( NULL == ( g_pD3D = Direct3DCreate8( D3D_SDK_VERSION ) ) )
    return E_FAIL;
```

The only parameter passed to **Direct3DCreate8** should always be D3D_SDK_VERSION. This informs Direct3D that the correct header files are being used. This value is incremented whenever a header or other change would require applications to be rebuilt. If the version does not match, **Direct3DCreate8** will fail.

The next step is to retrieve the current display mode by using the **IDirect3D8::GetAdapterDisplayMode** method as shown in the code fragment below.

```
D3DDISPLAYMODE d3ddm;
if( FAILED( g_pD3D->GetAdapterDisplayMode( D3DADAPTER_DEFAULT, &d3ddm ) ) )
    return E_FAIL;
```

The **Format** member of the **D3DDISPLAYMODE** structure will be used when creating the Direct3D device. To run in windowed mode, the **Format** member is used to create a back buffer that matches the adapter's current mode.

By filling in the fields of the **D3DPRESENT_PARAMETERS** you can specify how you want your 3-D application to behave. The CreateDevice sample project sets its **Windowed** member to TRUE, its **SwapEffect** member to D3DSWAPEFFECT_DISCARD, and its **BackBufferFormat** member to **d3ddm.Format**.

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory( &d3dpp, sizeof(d3dpp) );
d3dpp.Windowed   = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = d3ddm.Format;
```

The final step is to use the **IDirect3D8::CreateDevice** method to create the Direct3D device, as illustrated in the following code example.

```
if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                  &d3dpp, &g_pd3dDevice ) ) )
```

The preceding code sample creates the device with the default adapter by using the D3DADAPTER_DEFAULT flag. In most cases, the system will have only a single adapter, unless it has multiple graphics hardware cards installed. Indicate that you prefer a hardware device over a software device by specifying D3DDEVTYPE_HAL for the *DeviceType* parameter. This code sample uses D3DCREATE_SOFTWARE_VERTEXPROCESSING to tell the system to use software vertex processing. Note that if you tell the system to use hardware vertex processing by specifying D3DCREATE_HARDWARE_VERTEXPROCESSING,

you will see a significant performance gain on video cards that support hardware vertex processing.

Now that the Direct3D has been initialized, the next step is to ensure that you have a mechanism to process system messages, as described in Step 3: Handling System Messages.

Step 3: Handling System Messages

After you have created the application window and initialized Microsoft® Direct3D®, you are ready to render the scene. In most cases, Microsoft Windows® applications monitor system messages in their message loop, and they render frames whenever no messages are in the queue. However, the CreateDevice sample project waits until a WM_PAINT message is in the queue, telling the application that it needs to redraw all or part of its window.

```
// The message loop.
MSG msg;
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

Each time the loop runs, **DispatchMessage** calls **MsgProc**, which handles messages in the queue. When WM_PAINT is queued, the application calls **Render**, the application-defined function that will redraw the window. Then the Microsoft Win32® function **ValidateRect** is called to validate the entire client area.

The sample code for the message-handling function is shown below.

```
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            PostQuitMessage( 0 );
            return 0;

        case WM_PAINT:
            Render();
            ValidateRect( hWnd, NULL );
            return 0;
    }

    return DefWindowProc( hWnd, msg, wParam, lParam );
}
```

Now that the application handles system messages, the next step is to render the display, as described in Step 4: Rendering and Displaying a Scene.

Step 4: Rendering and Displaying a Scene

To render and display the scene, the sample code in this step clears the back buffer to a blue color, transfers the contents of the back buffer to the front buffer, and presents the front buffer to the screen.

To clear a scene, call the **IDirect3DDevice8::Clear** method.

```
// Clear the back buffer to a blue color
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 1.0f, 0 );
```

The first two parameters accepted by **Clear** inform Microsoft® Direct3D® of the size and address of the array of rectangles to be cleared. The array of rectangles describes the areas on the render target surface to be cleared.

In most cases, you use a single rectangle that covers the entire rendering target. You do this by setting the first parameter to 0 and the second parameter to NULL. The third parameter determines the method's behavior. You can specify a flag to clear a render-target surface, an associated depth buffer, the stencil buffer, or any combination of the three. This tutorial does not use a depth buffer, so D3DCLEAR_TARGET is the only flag used. The last three parameters are set to reflect clearing values for the render target, depth buffer, and stencil buffer. The CreateDevice sample project sets the clear color for the render target surface to blue (D3DCOLOR_XRGB(0,0,255)). The final two parameters are ignored by the **Clear** method because the corresponding flags are not present.

After clearing the viewport, the CreateDevice sample project informs Direct3D that rendering will begin, then it signals that rendering is complete, as shown in the following code fragment.

```
// Begin the scene.
g_pd3dDevice->BeginScene();

// Rendering of scene objects happens here.

// End the scene.
g_pd3dDevice->EndScene();
```

The **IDirect3DDevice8::BeginScene** and **IDirect3DDevice8::EndScene** methods signal to the system when rendering is beginning or is complete. You can call rendering methods only between calls to these methods. Even if rendering methods fail, you should call **EndScene** before calling **BeginScene** again.

After rendering the scene, you display it by using the **IDirect3DDevice8::Present** method.

```
g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```

The first two parameters accepted by **Present** are a source rectangle and destination rectangle. The sample code in this step presents the entire back buffer to the front buffer by setting these two parameters to NULL. The third parameter sets the destination window for this presentation. Because this parameter is set to NULL, the **hWndDeviceWindow** member of **D3DPRESENT_PARAMETERS** is used. The fourth parameter is the *DirtyRegion* parameter and in most cases should be set to NULL.

The final step for this tutorial is shutting down the application, as described in Step 5: Shutting Down

Step 5: Shutting Down

At some point during execution, your application must shut down. Shutting down a Microsoft® DirectX® application not only means that you destroy the application window, but you also deallocate any DirectX objects your application uses, and you invalidate the pointers to them. The CreateDevice sample project calls **Cleanup**, an application-defined function to handle this when it receives a WM_DESTROY message.

```
VOID Cleanup()
{
    if( g_pd3dDevice != NULL)
        g_pd3dDevice->Release();
    if( g_pD3D != NULL)
        g_pD3D->Release();
}
```

The preceding function deallocates the DirectX objects it uses by calling the **IUnknown::Release** methods for each object. Because this tutorial follows COM rules, the reference count for most objects should become zero and should be automatically removed from memory.

In addition to shutdown, there are times during normal execution—such as when the user changes the desktop resolution or color depth—when you might need to destroy and re-create the Microsoft Direct3D® objects in use. Therefore it is a good idea to keep your application's cleanup code in one place, which can be called when the need arises.

This tutorial has shown you how to create a device. Tutorial 2: Rendering Vertices shows you how to use vertices to draw geometric shapes.

Tutorial 2: Rendering Vertices

Applications written in Microsoft® Direct3D® use vertices to draw geometric shapes. Each 3-D scene includes one or more of these geometric shapes. The Vertices sample project creates the simplest shape, a triangle, and renders it to the display.

This tutorial shows how to use vertices to create a triangle with the following steps:

- Step 1: Defining a Custom Vertex Type
- Step 2: Setting Up the Vertex Buffer
- Step 3: Rendering the Display

Note

The path of the Vertices sample project is:
(SDK Root)\Samples\Multimedia\Direct3D\Tutorials\Tut02_Vertices.
The sample code in the Vertices project is nearly identical to the sample code in the CreateDevice project. The Rendering Vertices tutorial focuses only on the code unique to vertices, and does not cover initializing Direct3D, handling Microsoft Windows® messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.

Step 1: Defining a Custom Vertex Type

The Vertices sample project renders a 2-D triangle by using three vertices. This introduces the concept of the vertex buffer, which is a Microsoft® Direct3D® object that is used to store and render vertices. Vertices can be defined in many ways by specifying a custom vertex structure and corresponding custom flexible vector format (FVF). The format of the vertices in the Vertices sample project is shown in the following code fragment.

```
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex.
    DWORD color;        // The vertex color.
};
```

The structure above specifies the format of the custom vertex type. The next step is to define the FVF that describes the contents of the vertices in the vertex buffer. The following code fragment defines a FVF that corresponds with the custom vertex type created above.

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
```

Flexible vertex format flags describe what type of custom vertex is being used. The sample code above uses the D3DFVF_XYZRHW and D3DFVF_DIFFUSE flags, which tell the vertex buffer that the custom vertex type has a transformed point followed by a color component.

Now that the custom vector format and FVF are specified, the next step is to fill the vertex buffer with vertices, as described in Step 2: Setting Up the Vertex Buffer.

Note

The vertices in the Vertices sample project are transformed. In other words, they are already in 2-D window coordinates. This means that the point (0,0) is at the top-left

corner and the positive x-axis is right and the positive y-axis is down. These vertices are also lit, meaning that they are not using Direct3D lighting but are supplying their own color.

Step 2: Setting Up the Vertex Buffer

Now that the custom vertex format is defined, it is time to initialize the vertices. The Vertices sample project does this by calling the application-defined function **InitVB** after creating the required Microsoft® Direct3D® objects. The following code fragment initializes the values for three custom vertices.

```
CUSTOMVERTEX g_Vertices[] =
{
    { 150.0f,  50.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color
    { 250.0f, 250.0f, 0.5f, 1.0f, 0xff00ff00, },
    {  50.0f, 250.0f, 0.5f, 1.0f, 0xff00ffff, },
};
```

The preceding code fragment fills three vertices with the points of a triangle and specifies which color each vertex will emit. The first point is at (150, 50) and emits the color red (0xffff0000). The second point is at (250, 250) and emits the color green (0xff00ff00). The third point is at (50, 250) and emits the color blue-green (0xff00ffff). Each of these points has a depth value of 0.5 and an RHW of 1.0. For more information on this vector format see **Transformed and Lit Vertices**

The next step is to call **IDirect3DDevice8::CreateVertexBuffer** to create a vertex buffer as shown in the following code fragment.

```
if( FAILED( g_pd3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
                                              0 /* Usage */, D3DFVF_CUSTOMVERTEX,
                                              D3DPOOL_DEFAULT, &g_pVB ) ) )
    return E_FAIL;
```

The first two parameters of **CreateVertexBuffer** tell Direct3D the desired size and usage for the new vertex buffer. The next two parameters specify the vector format and memory location for the new buffer. The vector format here is D3DFVF_CUSTOMVERTEX, which is the FVF that the sample code specified earlier. The D3DPOOL_DEFAULT flag tells Direct3D to create the vertex buffer in the memory allocation that is most appropriate for this buffer. The final parameter is the address of the vertex buffer to create.

After creating a vertex buffer, it is filled with data from the custom vertices as shown in the following code fragment.

```
VOID* pVertices;
if( FAILED( g_pVB->Lock( 0, sizeof(g_Vertices), (BYTE**)&pVertices, 0 ) ) )
    return E_FAIL;
memcpy( pVertices, g_Vertices, sizeof(g_Vertices) );
g_pVB->Unlock();
```

The vertex buffer is first locked by calling **IDirect3DVertexBuffer8::Lock**. The first parameter is the offset into the vertex data to lock, in bytes. The second parameter is the size of the vertex data to lock, in bytes. The third parameter is the address of a BYTE pointer, filled with a pointer to vertex data. The fourth parameter tells the vertex buffer how to lock the data.

The vertices are then copied into the vertex buffer using **memcpy**. After the vertices are in the vertex buffer, a call is made to **IDirect3DVertexBuffer8::Unlock** to unlock the vertex buffer. This mechanism of locking and unlocking is required because the vertex buffer may be in device memory.

Now that the vertex buffer is filled with the vertices, it is time to render the display, as described in Step 3: Rendering the Display.

Step 3: Rendering the Display

Now that the vertex buffer is filled with vertices, it is time to render the display. Rendering the display starts by clearing the back buffer to a blue color and then calling **BeginScene**.

```
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 1.0f, 0L );
g_pd3dDevice->BeginScene();
```

Rendering vertex data from a vertex buffer requires a few steps. First, you need to set the stream source; in this case, use stream 0. The source of the stream is specified by calling **IDirect3DDevice8::SetStreamSource**.

```
g_pd3dDevice->SetStreamSource( 0, g_pVB, sizeof(CUSTOMVERTEX) );
```

The first parameter of **SetStreamSource** tells Microsoft® Direct3D® the source of the device data stream. The second parameter is the vertex buffer to bind to the data stream. The third parameter is the size of the component, in bytes. In the sample code above, the size of a CUSTOMVERTEX is used for the size of the component.

The next step is to let Direct3D know what vertex shader to use by calling **IDirect3DDevice8::SetVertexShader**. Full, custom vertex shaders are an advanced topic, but in most cases the vertex shader is only the FVF code. This lets Direct3D know what types of vertices it is dealing with. The following code fragment sets the vertex shader.

```
g_pd3dDevice->SetVertexShader( D3DFVF_CUSTOMVERTEX );
```

The only parameter for **SetVertexShader** is a handle to the vertex shader to use. The value for this parameter can be a handle returned by **IDirect3DDevice8::CreateVertexShader**, or an FVF code. Here, the FVF code defined by D3DFVF_CUSTOMVERTEX is used.

For more information on vertex shaders, see Vertex shaders.

The next step is to use **IDirect3DDevice8::DrawPrimitive** to render the vertices in the vertex buffer as shown in the following code fragment.

```
g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
```

The first parameter accepted by **DrawPrimitive** is a flag that tells Direct3D what type of primitives to draw. This sample uses the flag D3DPT_TRIANGLELIST to specify a list of triangles. The second parameter is the index of the first vertex to load. The third parameter tells the number of primitives to draw. Because this sample draws only one triangle, this value is set to 1.

For more information on different kinds of primitives, see **3-D Primitives**.

The last steps are to end the scene and then present the back buffer to the front buffer. This is shown in the following code fragment.

```
g_pd3dDevice->EndScene();
g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
```

After the back buffer is presented to the front buffer, the client window shows a triangle with three different colored points.

This tutorial has shown you how to use vertices to render geometric shapes. Tutorial 3: Using Matrices introduces the concept of matrices and how to use them.

Tutorial 3: Using Matrices

This tutorial introduces the concept of matrices and shows how to use them. The Vertices sample project rendered 2-D vertices to draw a triangle. However, in this tutorial you will be working with transformations of vertices in 3-D. Matrices are also used to set up cameras and viewports. For more information on working with transformation matrices, see 3-D Transformations.

Before the Matrices sample project renders geometry, it calls the **SetupMatrices** application-defined function to create and set the matrix transformations that are used to render the 3-D triangle. Typically, three types of transformation are set for a 3-D scene. Steps for creating each one of these typical transformations are listed below.

- Step 1: Defining the World Transformation Matrix
- Step 2: Defining the View Transformation Matrix
- Step 3: Defining the Projection Transformation Matrix

For more information on matrices and transformations, see **Matrices**.

For more information on 3-D Transformations, see **3-D Transformations**.

Note

The path of the Matrices sample project is:
(SDK Root)\Samples\Multimedia\Direct3D\Tutorials\Tut03_Matrices.

The order in which these transformation matrices are created does not affect the layout of the objects in a scene. However, Direct3D applies the matrices to the scene in the following order: (1) World ,(2) View, (3) Projection.
The sample code in the Matrices project is nearly identical to the sample code in the Vertices project. The Using Matrices tutorial focuses only on the code unique to matrices and does not cover initializing Direct3D, handling Microsoft Windows® messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.
This tutorial uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.

Step 1: Defining the World Transformation Matrix

The world transformation matrix defines how to translate, scale, and rotate the geometry in the 3-D model space.

The following code fragment rotates the triangle on the y-axis and then sets the current world transformation for the Microsoft® Direct3D® device.

```
D3DXMATRIX matWorld;
D3DXMatrixRotationY( &matWorld, timeGetTime()/150.0f );
g_pd3dDevice->SetTransform( D3DTS_WORLD, &matWorld );
```

The first step is to rotate the triangle around the Y-axis by calling the **D3DXMatrixRotationY** method. The first parameter is a pointer to a **D3DXMATRIX** structure that is the result of the operation. The second parameter is the angle of rotation in radians.

The next step is to call **IDirect3DDevice8::SetTransform** to set the world transformation for the Direct3D device. The first parameter accepted by **SetTransform** tells Direct3D which transformation to set. This sample uses the **D3DTS_WORLD** macro to specify that the world transformation should be set. The second parameter is a pointer to a matrix that is set as the current transformation.

For more information on world transformations, see **The World Transformation**.

After defining the world transformation for the scene, you can prepare the view transformation matrix. Again, note that the order in which transformations are defined is not critical. However, Direct3D applies the matrices to the scene in the following order: (1) World ,(2) View, (3) Projection.

Defining the view transformation matrix is described in Step 2: Defining the View Transformation Matrix.

Step 2: Defining the View Transformation Matrix

The view transformation matrix defines the position and rotation of the view. The view matrix is the camera for the scene.

The following code fragment creates the view transformation matrix and then sets the current view transformation for the Microsoft® Direct3D® device.

```
D3DXMATRIX matView;
D3DXMatrixLookAtLH( &matView, &D3DXVECTOR3( 0.0f, 3.0f,-5.0f ),
                    &D3DXVECTOR3( 0.0f, 0.0f, 0.0f ),
                    &D3DXVECTOR3( 0.0f, 1.0f, 0.0f ) );
g_pd3dDevice->SetTransform( D3DTS_VIEW, &matView );
```

The first step is to define the view matrix by calling **D3DXMatrixLookAtLH**. The first parameter is a pointer to a **D3DXMATRIX** structure that is the result of the operation. The second, third, and fourth parameters define the eye point, look-at point, and "up" direction. Here the eye is set back along the z-axis by five units and up three units, the look-at point is set at the origin, and "up" is defined as the y-direction.

The next step is to call **IDirect3DDevice8::SetTransform** to set the view transformation for the Direct3D device. The first parameter accepted by **SetTransform** tells Direct3D which transformation to set. This sample uses the D3DTS_VIEW flag to specify that the view transformation should be set. The second parameter is a pointer to a matrix that is set as the current transformation.

For more information on view transformations, see **The View Transformation**.

After defining the world transformation for the scene, you can prepare the projection transformation matrix. Again, note that the order in which transformations are defined is not critical. However, Direct3D applies the matrices to the scene in the following order: (1) World ,(2) View, (3) Projection.

Defining the projection transformation matrix is described in Step 3: Defining the Projection Transformation Matrix

Step 3: Defining the Projection Transformation Matrix

The projection transformation matrix defines how geometry is transformed from 3-D view space to 2-D viewport space.

The following code fragment creates the projection transformation matrix and then sets the current projection transformation for the Microsoft® Direct3D® device.

```
D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );
```

The first step is to call **D3DXMatrixPerspectiveFovLH** to set up the projection matrix. The first parameter is a pointer to a **D3DXMATRIX** structure that is the result of the operation. The second parameter defines the field of view, which tells how objects in the distance get smaller. A typical field of view is 1/4 pi, which is what the sample uses. The third parameter defines the aspect ratio. The sample uses the typical aspect ratio of 1. The fourth and fifth parameters define the near and far clipping plane. This determines the distance at which geometry should no longer be rendered. The Matrices sample project has its near clipping plane set at 1 and its far clipping plane set at 100.

The next step is to call **IDirect3DDevice8::SetTransform** to apply the transformation to the Direct3D device. The first parameter accepted by **SetTransform** tells Direct3D which transformation to set. This sample uses the D3DTS_PROJECTION flag to specify that the projection transformation should be set. The second parameter is a pointer to a matrix that is set as the current transformation.

For more information on projection transformations, see The Projection Transformation.

This tutorial has shown you how to use matrices. Tutorial 4: Creating and Using Lights shows how to add lights to your scene for more realism.

Tutorial 4: Creating and Using Lights

Microsoft® Direct3D® lights add more realism to 3-D objects. When used, each geometric object in the scene will be lit based on the location and type of lights that are used. The sample code in this tutorial introduces the topics of lights and materials.

This tutorial has the following steps to create a material and a light.

- Step 1: Initializing Scene Geometry
- Step 2: Setting up Material and Light

Note

The path of the Lights sample project is:
(SDK Root)\Samples\Multimedia\Direct3D\Tutorials\Tut04_Lights.
The sample code in the Lights project is nearly identical to the sample code in the Matrices project. The Creating and Using Lights tutorial focuses only on the code unique to creating and using lights and does not cover setting up Direct3D, handling Microsoft Windows messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.
This tutorial uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.
This tutorial makes use of matrices to transform geometry. For more information on matrices and transformations, see Tutorial 3: Using Matrices.

Step 1: Initializing Scene Geometry

One of the requirements of using lights is that each surface has a normal. To do this, the Lights sample project uses a different custom vertex type. The new custom vertex format has a 3-D position and a surface normal. The surface normal is used internally by Microsoft® Direct3D® for lighting calculations.

```
struct CUSTOMVERTEX
{
    D3DXVECTOR3 position; // The 3-D position for the vertex.
    D3DXVECTOR3 normal;   // The surface normal for the vertex.
};

// Custom FVF.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_NORMAL)
```

Now that the correct vector format is defined, the Lights sample project calls **InitGeometry**, an application-defined function that creates a cylinder. The first step is to create a vertex buffer that stores the points of the cylinder as shown in the following sample code.

```
// Create the vertex buffer.
if( FAILED( g_pd3dDevice->CreateVertexBuffer( 50*2*sizeof(CUSTOMVERTEX),
                                              0 /* Usage */, D3DFVF_CUSTOMVERTEX,
                                              D3DPOOL_DEFAULT, &g_pVB ) ) )
    return E_FAIL;
```

The next step is to fill the vertex buffer with the points of the cylinder. Note that in the following sample code, each point is defined by a position and a normal.

```
for( DWORD i=0; i<50; i++ )
{
    FLOAT theta = (2*D3DX_PI*i)/(50-1);
    pVertices[2*i+0].position = D3DXVECTOR3( sinf(theta),-1.0f, cosf(theta) );
    pVertices[2*i+0].normal   = D3DXVECTOR3( sinf(theta), 0.0f, cosf(theta) );
    pVertices[2*i+1].position = D3DXVECTOR3( sinf(theta), 1.0f, cosf(theta) );
    pVertices[2*i+1].normal   = D3DXVECTOR3( sinf(theta), 0.0f, cosf(theta) );
}
```

After the preceding sample code fills the vertex buffer with the vertices for a cylinder, the vertex buffer is ready for rendering. But first, the material and light for this scene must be set up before rendering the cylinder. This is described in Step 2: Setting up Material and Light.

Step 2: Setting up Material and Light

To use lighting in Microsoft® Direct3D®, you must create one or more lights. To determine which color a geometric object reflects, a material is created that is used to render geometric objects. Before rendering the scene, the Lights sample project calls **SetupLights**, an application-defined function that sets up one material and one directional light. This function takes the following steps to create a material and a light.

- Step 2.1: Creating a Material
- Step 2.2: Creating a Light

Step 2.1: Creating a Material

A material defines the color that is reflected off the surface of a geometric object when a light hits it. The following code fragment uses the **D3DMATERIAL8** structure to create a material that is yellow.

```
D3DMATERIAL8 mtrl;
ZeroMemory( &mtrl, sizeof(D3DMATERIAL8) );
mtrl.Diffuse.r = mtrl.Ambient.r = 1.0f;
mtrl.Diffuse.g = mtrl.Ambient.g = 1.0f;
mtrl.Diffuse.b = mtrl.Ambient.b = 0.0f;
mtrl.Diffuse.a = mtrl.Ambient.a = 1.0f;
g_pd3dDevice->SetMaterial( &mtrl );
```

The diffuse color and ambient color for the material are set to yellow. The call to the **IDirect3DDevice8::SetMaterial** method applies the material to the Microsoft® Direct3D® device used to render the scene. The only parameter that **SetMaterial** accepts is the address of the material to set. After this call is made, every primitive will be rendered with this material until another call is made to **SetMaterial** that specifies a different material.

Now that material has been applied to the scene, the next step is to create a light. This is described in Step 2.2: Creating a Light.

Step 2.2: Creating a Light

There are three types of lights available in Microsoft® Direct3D®: point lights, directional lights, and spotlights. The sample code creates a directional light, which is a light that goes in one direction, and it oscillates the direction of the light.

The following code fragment uses the D3DLIGHT8 structure to create a directional light.

```
D3DXVECTOR3 vecDir;
D3DLIGHT8 light;
ZeroMemory( &light, sizeof(D3DLIGHT8) );
light.Type       = D3DLIGHT_DIRECTIONAL;
```

The following code fragment sets the diffuse color for this light to white.

```
light.Diffuse.r  = 1.0f;
light.Diffuse.g  = 1.0f;
light.Diffuse.b  = 1.0f;
```

The following code fragment rotates the direction of the light around in a circle.

```
vecDir = D3DXVECTOR3(cosf(timeGetTime()/360.0f),
                     0.0f,
                     sinf(timeGetTime()/360.0f) );
D3DXVec3Normalize( (D3DXVECTOR3*)&light.Direction, &vecDir );
```

The call to **D3DXVec3Normalize** normalizes the direction vector used to determine the direction of the light.

A range can be specified to tell Direct3D how far the light will have an effect. This member does not affect directional lights. The following code fragment assigns a range of 1000 units to this light.

```
light.Range       = 1000.0f;
```

The following code fragment assigns the light to the Direct3D device by calling **IDirect3DDevice8::SetLight**.

```
g_pd3dDevice->SetLight( 0, &light );
```

The first parameter that **SetLight** accepts is the index that this light will be assigned to. Note that if a light already exists at that location, it will be overwritten by the new light. The second parameter is a pointer to the light structure that defines the light. The Lights sample project places this light at index 0.

The following code fragment enables the light by calling **IDirect3DDevice8::LightEnable**.

```
g_pd3dDevice->LightEnable( 0, TRUE);
```

The first parameter that **LightEnable** accepts is the index of the light to enable. The second parameter is a Boolean value that tells whether to turn the light on (TRUE) or off (FALSE). In the sample code above, the light at index 0 is turned on.

The following code fragment tells Direct3D to render lights by calling **IDirect3DDevice8::SetRenderState**.

```
g_pd3dDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
```

The first two parameters that **SetRenderState** accepts is which device state variable to modify and what value to set it to. This code sample sets the D3DRS_LIGHTING device variable to TRUE, which has the effect of enabling the rendering of lights.

The final step in this code sample is to turn on ambient lighting by again calling **SetRenderState**.

```
g_pd3dDevice->SetRenderState( D3DRS_AMBIENT, 0x00202020 );
```

The preceding code fragment sets the D3DRS_AMBIENT device variable to a light gray color (0x00202020). Ambient lighting will light up all objects by the given color.

For more information on lighting, see Mathematics of Direct3D Lighting.

This tutorial has shown you how to use lights and materials. Tutorial 5: Using Texture Maps shows you how to add texture to surfaces.

Tutorial 5: Using Texture Maps

While lights and materials add a great deal of realism to a scene, nothing adds more realism than adding textures to surfaces. Textures can be thought of as wallpaper that is shrink-wrapped onto a surface. You could place a wood texture on a cube to make it look like the cube is actually made of wood. The Texture sample project adds a banana peel texture to the cylinder created in Tutorial 4: Creating and Using Lights. This tutorial covers how to load textures, set up vertices, and display objects with texture.

This tutorial implements textures using the following steps:

- Step 1: Defining a Custom Vector Format
- Step 2: Initializing Screen Geometry
- Step 3: Rendering the Scene

Note

The path of the Texture sample project is:
(SDK Root)\Samples\Multimedia\Direct3D\Tutorials\Tut05_Textures.
The sample code in the Texture project is nearly identical to the sample code in the Lights project, except that the Texture sample project does not create a material or a light. The Using Texture Maps tutorial focuses only on the code unique to textures and does not cover initializing Microsoft® Direct3D®, handling Microsoft Windows® messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.
This tutorial uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.
This tutorial makes use of matrices to transform geometry. For more information on matrices and transformations, see Tutorial 3: Using Matrices.

Step 1: Defining a Custom Vector Format

Before using textures, a custom vertex format that includes texture coordinates must be used. Texture coordinates tell Microsoft® Direct3D® where to place a texture for

each vector in a primitive. Texture coordinates range from 0.0 to 1.0, where (0.0, 0.0) represents the top-left side of the texture and (1.0, 1.0) represents the bottom-right side of the texture.

The following sample code shows how the Texture sample project sets up its custom vertex format to include texture coordinates.

```
struct CUSTOMVERTEX
{
    D3DXVECTOR3 position; // The position.
    D3DCOLOR    color;    // The color.
    FLOAT       tu, tv;   // The texture coordinates.
};

// The custom FVF, which describes the custom vertex structure.
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1)
```

For more information on texture coordinates, see **Texture Coordinates**.

Now that a custom vertex type has been defined, the next step is to load a texture and create a cylinder, as described in Step 2: Initializing Screen Geometry.

Step 2: Initializing Screen Geometry

Before rendering, the Texture sample project calls **InitGeometry**, an application-defined function that creates a texture and initializes the geometry for a cylinder.

Textures are created from file-based images. The following code fragment uses **D3DXCreateTextureFromFile** to create a texture from Banana.bmp that will be used to cover the surface of the cylinder.

```
if( FAILED( D3DXCreateTextureFromFile( g_pd3dDevice, "Banana.bmp",
                                       &g_pTexture ) ) )
    return E_FAIL;
```

The first parameter that **D3DXCreateTextureFromFile** accepts is a pointer to the Microsoft® Direct3D® device that will be used to render the texture. The second parameter is a pointer to an ANSI string that specifies the filename from which to create the texture. This sample specifies Banana.bmp to load the image from that file. The third parameter is the address of a pointer to a texture object.

When the banana texture is loaded and ready to use, the next step is to create the cylinder. The following code sample fills the vertex buffer with a cylinder. Note that each point has the texture coordinates (tu, tv).

```
for( DWORD i=0; i<50; i++ )
{
    FLOAT theta = (2*D3DX_PI*i)/(50-1);

    pVertices[2*i+0].position = D3DXVECTOR3( sinf(theta),-1.0, cosf(theta) );
```

```
    pVertices[2*i+0].color    = 0xffffffff;
    pVertices[2*i+0].tu       = ((FLOAT)i)/(50-1);
    pVertices[2*i+0].tv       = 1.0f;

    pVertices[2*i+1].position = D3DXVECTOR3( sinf(theta), 1.0, cosf(theta) );
    pVertices[2*i+1].color    = 0xff808080;
    pVertices[2*i+1].tu       = ((FLOAT)i)/(50-1);
    pVertices[2*i+1].tv       = 0.0f;
}
```

Each vertex includes position, color, and texture coordinates. The code sample above sets the texture coordinates for each point so that the texture will wrap smoothly around the cylinder.

Now that the texture is loaded and the vertex buffer is ready for rendering, it is time to render the display, as described in Step 3: Rendering the Scene.

Step 3: Rendering the Scene

After scene geometry has been initialized, it is time to render the scene. In order to render an object with texture, the texture must be set as one of the current textures. The next step is to set the texture stage states values. Texture stage states enable you to define the behavior of how a texture or textures are to be rendered. For example, you could blend multiple textures together.

The Texture sample project starts by setting the texture to use. The following code fragment sets the texture that the Microsoft® Direct3D® device will use to render with **IDirect3DDevice8::SetTexture**.

```
g_pd3dDevice->SetTexture( 0, g_pTexture );
```

The first parameter that **SetTexture** accepts is a stage identifier to set the texture to. A device can have up to eight set textures, so the maximum value here is 7. The Texture sample project has only one texture and places it at stage 0. The second parameter is a pointer to a texture object. The Texture sample project uses the texture object that it created in its **InitGeometry** application-defined function.

The following code sample sets the texture stage state values by calling the **IDirect3DDevice8::SetTextureStageState** method.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_MODULATE );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAOP,   D3DTOP_DISABLE );
```

The first parameter that **SetTextureState** accepts is the stage index for the state variable to be set. This code sample is changing the values for the texture at stage 0, so it puts a 0 here. The next parameter is the texture state to set. For a list of all valid texture states and their meaning, see **D3DTEXTURESTAGESTATETYPE**. The

next parameter is the value to set the texture state to. The value that you place here is based on the texture stage state value that you are modifying.

After setting up the desired values for each texture stage state, the cylinder can be rendered and texture will be added to the surface.

Another way to use texture coordinates is to have them automatically generated. This is done by using a texture coordinate index (TCI). The TCI uses a texture matrix to transform the (x,y,z) TCI coordinates into (tu, tv) texture coordinates. In the Texture sample project, the position of the vertex in camera space is used to generate texture coordinates.

The first step is to create the matrix that will be used for the transformation, as demonstrated in the following code fragment.

```
D3DXMATRIX mat;
mat._11 = 0.25f; mat._12 = 0.00f; mat._13 = 0.00f; mat._14 = 0.00f;
mat._21 = 0.00f; mat._22 =-0.25f; mat._23 = 0.00f; mat._24 = 0.00f;
mat._31 = 0.00f; mat._32 = 0.00f; mat._33 = 1.00f; mat._34 = 0.00f;
mat._41 = 0.50f; mat._42 = 0.50f; mat._43 = 0.00f; mat._44 = 1.00f;
```

After the matrix is created, it must be set by calling **IDirect3DDevice8::SetTransform**, as shown in the following code fragment.

```
g_pd3dDevice->SetTransform( D3DTS_TEXTURE0, &mat );
```

The D3DTS_TEXTURE0 flag tells Direct3D to apply the transformation to the texture located at texture stage 0. The next step that this sample takes is to set more texture stage state values to get the desired effect. That is done in the following code fragment.

```
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTFF_COUNT2 );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEPOSITION );
```

The texture coordinates are set up, and now the scene is ready to be rendered. Notice that the coordinates are automatically created for the cylinder. This particular setup gives the effect of the texture being laid over the rendering screen after the geometric shapes have been rendered.

For more information on textures, see Textures.

This tutorial has shown you how to use textures for surfaces. Tutorial 6: Using Meshes shows you how to use complex geometric shapes with meshes.

Tutorial 6: Using Meshes

Complicated geometry is usually modeled using 3-D modeling software and saved to a file. An example of this is the .x file format. Microsoft® Direct3D® uses meshes to

load the objects from these files. Meshes are somewhat complicated, but D3DX contains functions that make using meshes easier. The Meshes sample project introduces the topic of meshes and shows how to load, render, and unload a mesh.

This tutorial shows how to load, render, and unload a mesh using the following steps.

- Step 1: Loading a Mesh Object
- Step 2: Rendering a Mesh Object
- Step 3: Unloading a Mesh Object

Notes

The path of the Meshes sample project is:
(SDK Root)\Samples\Multimedia\Direct3D\Tutorials\Tut06_Meshes.

The sample code in the Meshes project is nearly identical to the sample code in the Lights project, except that the code in the Meshes project does not create a material or a light. The Using Meshes tutorial focuses only on the code unique to meshes and does not cover setting up Direct3D, handling Microsoft Windows® messages, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.

This tutorial uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.

This tutorial makes use of matrices to transform geometry. For more information on matrices and transformations, see Tutorial 3: Using Matrices.

This tutorial uses textures to cover the surface of the mesh. For more information on loading and using textures, see Tutorial 5: Using Texture Maps.

Step 1: Loading a Mesh Object

A Microsoft® Direct3D® application must first load a mesh before using it. The Meshes sample project loads the tiger mesh by calling **InitGeometry**, an application-defined function, after loading the required Direct3D objects.

A mesh needs a material buffer that will store all the materials and textures that will be used. The function starts by declaring a material buffer as shown in the following code fragment.

```
LPD3DXBUFFER pD3DXMtrlBuffer;
```

The following code fragment uses the **D3DXLoadMeshFromX** method to load the mesh.

```
// Load the mesh from the specified file.
if( FAILED( D3DXLoadMeshFromX( "tiger.x", D3DXMESH_SYSTEMMEM,
                    g_pd3dDevice, NULL,
                    &pD3DXMtrlBuffer, &g_dwNumMaterials,
                    &g_pMesh ) ) )
    return E_FAIL;
```

The first parameter that **D3DXLoadMeshFromX** accepts is a pointer to a string that tells the name of the Microsoft DirectX® file to load. This sample loads the tiger mesh from Tiger.x.

The second parameter tells Direct3D how to create the mesh. The sample uses the D3DXMESH_SYSTEMMEM flag, which is equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM. Both of these flags tell Direct3D to put the index buffer and vertex buffer for the mesh in system memory.

The third parameter is a pointer to a Direct3D device that will be used to render the mesh.

The fourth parameter is a pointer to an **ID3DXBuffer** object. This object will be filled with information about neighbors for each face. This information is not required for this sample, so this parameter is set to NULL.

The fifth parameter also takes a pointer to a **ID3DXBuffer** object. After this method is finished, this object will be filled with D3DXMATERIAL structures for the mesh.

The sixth parameter is a pointer to the number of D3DXMATERIAL structures placed into the *ppMaterials* array after the method returns.

The seventh parameter is the address of a pointer to a mesh object, representing the loaded mesh.

After loading the mesh object and material information, you need to extract the material properties and texture names from the material buffer.

The Meshes sample project does this by first getting the pointer to the material buffer. The following code fragment uses the **ID3DXBuffer::GetBufferPointer** method to get this pointer.

```
D3DXMATERIAL* d3dxMaterials = (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
```

The following code fragment creates new mesh and texture objects based on the total number of materials for the mesh.

```
g_pMeshMaterials = new D3DMATERIAL8[g_dwNumMaterials];
g_pMeshTextures  = new LPDIRECT3DTEXTURE8[g_dwNumMaterials];
```

For each material in the mesh the following steps occur.

The first step is to copy the material, as shown in the following code fragment.

```
g_pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
```

The second step is to set the ambient color for the material, as shown in the following code fragment.

```
g_pMeshMaterials[i].Ambient = g_pMeshMaterials[i].Diffuse;
```

The final step is to create the texture for the material, as shown in the following code fragment.

```
// Create the texture.
if( FAILED( D3DXCreateTextureFromFile( g_pd3dDevice,
                                       d3dxMaterials[i].pTextureFilename,
                                       &g_pMeshTextures[i] ) ) )
    g_pMeshTextures[i] = NULL;
}
```

After loading each material, you are finished with the material buffer and need to release it by calling **IUnknown::Release**.

```
pD3DXMtrlBuffer->Release();
```

The mesh, along with the corresponding materials and textures are loaded. The mesh is ready to be rendered to the display, as described in Step 2: Rendering a Mesh Object.

Step 2: Rendering a Mesh Object

In step 1 the mesh was loaded and is now ready to be rendered. It is divided into a subset for each material that was loaded for the mesh. To render each subset, the mesh is rendered in a loop. The first step in the loop is to set the material for the subset, as shown in the following code fragment.

```
g_pd3dDevice->SetMaterial( &g_pMeshMaterials[i] );
```

The second step in the loop is to set the texture for the subset, as shown in the following code fragment.

```
g_pd3dDevice->SetTexture( 0, g_pMeshTextures[i] );
```

After setting the material and texture, the subset is drawn with the **ID3DXBaseMesh::DrawSubset** method, as shown in the following code fragment.

```
g_pMesh->DrawSubset( i );
```

The **DrawSubset** method takes a **DWORD** that specifies which subset of the mesh to draw. This sample uses a value that is incremented each time the loop runs.

After using a mesh, it is important to properly remove the mesh from memory, as described in Step 3: Unloading a Mesh Object.

Step 3: Unloading a Mesh Object

After any Microsoft® DirectX® program finishes, it needs to deallocate any DirectX objects that it used and invalidate the pointers to them. The mesh objects used in this

sample also need to be deallocated. When it receives a WM_DESTROY message, the Meshes sample project calls **Cleanup**, an application-defined function, to handle this.

The following code fragment deletes the material list.

```
if( g_pMeshMaterials )
    delete[] g_pMeshMaterials;
```

The following code fragment deallocates each individual texture that was loaded and then deletes the texture list.

```
if( g_pMeshTextures )
{
    for( DWORD i = 0; i < g_dwNumMaterials; i++ )
    {
        if( g_pMeshTextures[i] )
            g_pMeshTextures[i]->Release();
    }
delete[] g_pMeshTextures;
```

The following code fragment deallocates the mesh object.

```
Delete the mesh object
if( g_pMesh )
    g_pMesh->Release();
```

This tutorial has shown you how to load and render meshes. This is the last tutorial in this section. To see how a typical Direct3D application is written, see DirectX Graphics C/C++ Samples.

DirectX Graphics Visual Basic Tutorials

The tutorials in this section show how to use Microsoft® Direct3D® and Direct3DX in a Microsoft Visual Basic® application for common tasks by dividing those tasks into required steps. In some cases, steps are organized into substeps for clarity. The following tutorials are presented.

- Tutorial 1: Creating a Device
- Tutorial 2: Rendering Vertices
- Tutorial 3: Using Matrices
- Tutorial 4: Creating and Using Lights
- Tutorial 5: Using Texture Maps
- Tutorial 6: Using Meshes

Note

The sample code in these tutorials is from source projects whose location is provided in each tutorial.
Some comments in the included sample code might differ from the source files in the SDK. Changes are made for brevity only, and are limited to comments to avoid changing the behavior of the code.

See Also

DirectX Graphics Visual Basic Samples

Tutorial 1: Creating a Device

To use Microsoft® Direct3D® in Microsoft Visual Basic®, you first create a form for your application window, then create and initialize Direct3D objects. You use these objects through their defined methods, which are also used to create other objects required to render a scene. The CreateDevice sample project upon which this tutorial is based illustrates these tasks by creating a Direct3D device and rendering a blue screen. This tutorial uses the following steps to create a form, set up Direct3D, render a scene, and eventually shut down.

- Step 1: Creating a Form
- Step 2: Initializing Direct3D
- Step 3: Rendering and Displaying a Scene
- Step 4: Shutting Down

Note

The path of the CreateDevice sample project is:
(SDK Root)
\Samples\Multimedia\VBSamples\Direct3D\Tutorials\Tut01_CreateDevice.

Step 1: Creating a Form

A Microsoft® Visual Basic® form serves as your application window. Set the properties for your form in the following manner:

```
BorderStyle    =  2 - Sizeable
Caption        =  "Create Device"
Height         =  3600
Left           =  0
Top            =  0
Width          =  4800
LinkTopic      =  "Form1"
MaxButton      =  True
MinButton      =  True
```

```
ScaleHeight    =   3195
ScaleMode      =   1 - Twip
ScaleWidth     =   254
StartUpPosition =   3 - Windows Default
```

To display your application window, include the following line in your **Form_Load** procedure:

```
Me.Show
```

With the application window ready, you can begin initializing the main Microsoft DirectX® objects, as described in Step 2: Initializing Direct3D.

Step 2: Initializing Direct3D

The CreateDevice sample project performs system initialization in its InitD3D application-defined function called from **Form_Load** after the form is shown. After you display a form for the application, you are ready to initialize the Microsoft® Direct3D® device that you will use to render the scene. This process includes creating a Direct3D object, setting the presentation parameters, and finally creating the Direct3D device.

To create a Direct3D object, use the **DirectX8.Direct3DCreate** method. You can use this Direct3D object to enumerate devices, types, modes, and so on.

```
Set g_D3D = g_DX.Direct3DCreate()
If g_D3D Is Nothing Then Exit Function
```

The next step is to retrieve the current display mode by using the **IDirect3D8.GetAdapterDisplayMode** method as shown in the code fragment below.

```
Dim Mode As D3DDISPLAYMODE
g_D3D.GetAdapterDisplayMode D3DADAPTER_DEFAULT, mode
```

The **Format** member of the **D3DDISPLAYMODE** structure will be used when creating the Direct3D device. To run in windowed mode, the **Format** member is used to create a back buffer that matches the adapter's current mode.

By filling in the fields of the **D3DPRESENT_PARAMETERS** object, you can specify how you want your 3-D application to behave. The CreateDevice sample project sets its **Windowed** member to 1(TRUE), its **SwapEffect** member to D3DSWAPEFFECT_COPY_VSYNC, and its **BackBufferFormat** member to **mode.Format**.

```
Dim d3dpp As D3DPRESENT_PARAMETERS
d3dpp.Windowed = 1
d3dpp.SwapEffect = D3DSWAPEFFECT_COPY_VSYNC
d3dpp.BackBufferFormat = mode.Format
```

The final step is to use **Direct3D8.CreateDevice** to create the Direct3D device, as illustrated in the following code example.

```
Set g_D3DDevice = g_D3D.CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
hWnd, _
                                D3DCREATE_SOFTWARE_VERTEXPROCESSING, d3dpp)
If g_D3DDevice Is Nothing Then Exit Function
```

The preceding code creates the device with the default adapter by using the D3DADAPTER_DEFAULT flag. In most cases, the system will have only a single adapter, unless it has multiple graphics hardware cards installed. Indicate that you prefer a hardware device over a software device by specifying D3DDEVTYPE_HAL for the *DeviceType* parameter. This sample uses D3DCREATE_SOFTWARE_VERTEXPROCESSING to tell the system to use software vertex processing. Note that if you tell the system to use hardware vertex processing by specifying D3DCREATE_HARDWARE_VERTEXPROCESSING, you will see a significant performance gain on video cards that support hardware vertex processing.

Now that the required Direct3D objects have been created, the next step, to execute the rendering loop, is described in Step 3: Rendering and Displaying a Scene.

Step 3: Rendering and Displaying a Scene

The CreateDevice sample project uses a timer to render the screen at a given interval. The sample sets the timer's *Interval* property to 40. This means that every forty milliseconds the function **Timer1_Timer** will fire. When the timer fires, it calls the application-defined function **Render**, which renders and displays the scene. The CreateDevice sample project clears the back buffer to a blue color, transfers the contents of the back buffer to the front buffer, and presents the front buffer to the screen.

To render a scene, start by calling the **Direct3DDevice8.Clear** method.

```
g_D3DDevice.Clear 0, ByVal 0, D3DCLEAR_TARGET, &HFF&, 1#, 0
```

The first two parameters accepted by **Clear** inform Microsoft® Direct3D® of the size and address of the array of rectangles to be cleared The array of rectangles describes the areas on the render target surface to be cleared.

In most cases, you will use a single rectangle that covers the entire rendering target. This is done by setting the first parameter to 0 and the second parameter to ByVal 0. The third parameter determines the method's behavior. You can specify a flag to clear a render-target surface, an associated depth buffer, the stencil buffer, or any combination of the three. This tutorial does not use a depth buffer, so D3DCLEAR_TARGET is the only flag used. The last three parameters are set to reflect clearing values for the render target, depth buffer, and stencil buffer. The CreateDevice sample project sets the clear color for the render target surface to blue

(&HFF&). The final two parameters are ignored by the **Clear** method because the corresponding flags are not present.

After clearing the viewport, the CreateDevice sample project informs Direct3D that rendering will begin, then signals that rendering is complete, as shown in the following code fragment.

```
' Begin the scene.
g_D3DDevice.BeginScene

' Rendering of scene objects occur here.

' End the scene.
g_D3DDevice.EndScene
```

The **Direct3DDevice8.BeginScene** and Direct3DDevice8.EndScene methods signal to the system when rendering is beginning or is complete. You can call rendering methods only between calls to these methods. Even if rendering methods fail, you should call **EndScene** before calling **BeginScene** again.

After rendering the scene, you display it by using the **Direct3DDevice8.Present** method.

```
g_D3DDevice.Present ByVal 0, ByVal 0, 0, ByVal 0
```

The first two parameters accepted by **Present** are a source rectangle and a destination rectangle. This sample presents the entire back buffer to the front buffer by setting these two parameters to ByVal 0. The third parameter sets the destination window for this presentation. Because this parameter is set to 0, the **hWndDeviceWindow** member of D3DPRESENT_PARAMETERS is used. The fourth parameter is the *DirtyRegion* parameter and in most cases should be set to ByVal 0.

The final step for this tutorial, shutting down the application, is described in Step 4: Shutting Down.

Step 4: Shutting Down

At some point during execution, your application must shut down. When shutting down a Microsoft® DirectX® application, you need to disassociate any DirectX objects your application used by setting them to Nothing. The CreateDevice sample project calls **Cleanup**, an application-defined function, to handle this cleanup. **Cleanup** is called from **Form_Unload** when the form is unloading.

```
Private Sub Form_Unload(Cancel As Integer)

    Set g_D3D = Nothing
    Set g_D3DDevice = Nothing

End Sub
```

The preceding function sets its Microsoft Direct3D® objects to Nothing, which in turn disassociates the variable from the actual object. In addition to shut down, there are times during normal execution—such as when the user changes the desktop resolution or color depth—when you might need to destroy and re-create the Direct3D objects in use. As a result, it's handy to keep your application's cleanup code in one place, which can be called when the need arises.

This tutorial has shown you how to create a device. Tutorial 2: Rendering Vertices shows you how to use vertices to draw geometric shapes.

Tutorial 2: Rendering Vertices

Applications written in Microsoft® Direct3D® use vertices to draw geometric shapes. Each 3-D scene inclides one or more of these geometric shapes. The Vertices sample project creates the simplest shape, a triangle, and renders it to the display. This tutorial covers how to select a custom vertex type, set up a vertex buffer, and render vertices to the display.

This tutorial shows how to use vertices to create a triangle with the following steps:

- Step 1: Defining a Custom Vertex Type
- Step 2: Setting Up the Vertex Buffer
- Step 3: Rendering the Display

Note

The path of the Vertices sample project is:
(SDK Root)
\Samples\Multimedia\VBSamples\Direct3D\Tutorials\Tut01_Vertices.
The code in the Vertices sample project is nearly identical to the code in CreateDevice sample project. This tutorial focuses only on the code unique to vertices and does not cover initializing Direct3D, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device

Step 1: Defining a Custom Vertex Type

The Vertices sample project renders a 2-D triangle by using three vertices. This introduces the concept of the vertex buffer, which is a Microsoft® Direct3D® object that is used to store and render vertices. Vertices can be defined many different ways by specifying a custom vertex structure and corresponding custom flexible vector format (FVF). The format of the vertices in the Vertices sample project is shown in the following code fragment.

```
Private Type CUSTOMVERTEX
    x As Single         'x in screen space.
    y As Single         'y in screen space.
    z As Single         'normalized z.
    rhw As Single       'normalized z rhw.
```

```
    color As Long       'vertex color.
End Type
```

The structure above specifies the format of the custom vertex type. The next step is to define the FVF that describes the contents of the vertices in the vertex buffer. The following code fragment defines an FVF that corresponds with the custom vertex type created above.

```
Const D3DFVF_CUSTOMVERTEX = (D3DFVF_XYZRHW Or D3DFVF_DIFFUSE)
```

The **Flexible Vertex Format Flags** describes what type of custom vertex is being used. The Vertices sample project uses the D3DFVF_XYZRHW and D3DFVF_DIFFUSE flags, which tells the vertex buffer that the custom vertex type has a transformed point followed by a color component.

Now that the custom vector format and FVF are specified, the next step is to fill the vertex buffer with vertices, as described in Step 2: Setting Up the Vertex Buffer.

Note

The vertices in the Vertices sample project are transformed, meaning that they are already in 2-D window coordinates. This means that the point (0,0) is at the top left hand corner and the positive x-axis is right and the positive y-axis is down. These vertices are also lit, which means that they are not using Direct3D lighting, but are supplying their own color.

Step 2: Setting Up the Vertex Buffer

Now that the custom vertex format is defined, it is time to initialize the vertices. The Vertices sample project does this by calling the application-defined function **InitVB** after creating the required Microsoft® Direct3D® objects. The code following fragment initializes the values for three custom vertices.

```
With Vertices(0): .x = 150: .y =  50: .z = 0.5: .rhw = 1: .color = &HFFFF0000: End With
With Vertices(1): .x = 250: .y = 250: .z = 0.5: .rhw = 1: .color = &HFF00FF00: End With
With Vertices(2): .x =  50: .y = 250: .z = 0.5: .rhw = 1: .color = &HFF00FFFF: End With
```

The preceding code fills three vertices with the points of a triangle and specifies which color each vertex will emit. The first point is at (150, 50) and emits the color red (&HFFFF0000). The second point is at (250, 250) and emits the color green (&HFF00FF00). The third point is at (50, 250) and emits the color blue-green (&HFF00FFFF). Each of these points has a pixel depth of 0.5 and an RHW of 1. For more information on this vector format see **Transformed and Lit Vertices**

The next step is to call **Direct3DDevice8.CreateVertexBuffer** to create a vertex buffer as shown in the following code fragment.

```
Set g_VB = g_D3DDevice.CreateVertexBuffer(VertexSizeInBytes * 3, _
           0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT)
If g_VB Is Nothing Then Exit Function
```

The first two parameters for **CreateVertexBuffer** tell Direct3D the desired size and usage for the new vertex buffer. The next two parameters specify the vector format and memory location for the new buffer. The vector format here is D3DFVF_CUSTOMVERTEX, which is the FVF that the Vertices sample project specified earlier. The D3DPOOL_DEFAULT flag tells Direct3D to create the vertex buffer in the memory that is most appropriate for this buffer. The final parameter is the address of the vertex buffer to create.

After creating a vertex buffer, it is filled with data from the custom vertices by calling the **D3DVertexBuffer8SetData** method as shown in the following code fragment.

```
D3DVertexBuffer8SetData g_VB, 0, VertexSizeInBytes * 3, 0, Vertices(0)
```

The first parameter that **D3DVertexBuffer8SetData** accepts is the vertex buffer to place the data in. The second parameter is offset, in bytes, from the start of the buffer where data is set. The third parameter is the size of the buffer, in bytes. The fourth flag is a combination of one or more flags that describe how to lock the buffer. The sample sets this parameter to 0, which tells the function to send default flags to the lock. The fifth parameter is the buffer that contains the data to place in the vertex buffer.

Now that the vertex buffer is filled with the vertices, it is time to render the display, as described in Step 3: Rendering the Display.

Step 3: Rendering the Display

Now that the vertex buffer is filled with vertices, it is time to render the display. Rendering the display starts by clearing the back buffer to a blue color and then calling **BeginScene**.

```
g_D3DDevice.Clear 0, ByVal 0, D3DCLEAR_TARGET, &HFF&, 1#, 0
g_D3DDevice.BeginScene
```

Rendering vertex data from a vertex buffer is divided into a few steps. First, you need to set the stream source; in this case, use stream 0. The source of the stream is specified by calling **Direct3DDevice8.SetStreamSource**.

```
g_D3DDevice.SetStreamSource 0, g_VB, sizeOfVertex
```

The first parameter of **SetStreamSource** tells Microsoft® Direct3D® the source of the device data stream. The second parameter is the vertex buffer to bind to the data stream. The third parameter is the size of the component, in bytes.

The next step is to let Direct3D know what vertex shader to use by calling **Direct3DDevice8.SetVertexShader**. Full, custom vertex shaders are an advanced topic, but in most cases the vertex shader is just the FVF. This lets Direct3D know what types of vertices it is dealing with. The following code fragment sets the vertex shader.

```
g_D3DDevice.SetVertexShader D3DFVF_CUSTOMVERTEX
```

The only parameter for **SetVertexShader** is a handle to the vertex shader to use. The value for this parameter can be a handle returned by **Direct3DDevice8.CreateVertexShader**, or a FVF code. Here, the FVF code defined by D3DFVF_CUSTOMVERTEX is used.

For more information on vertex shaders, see **Vertex Shaders**.

The next step is to use **Direct3DDevice8.DrawPrimitive** to render the vertices in the vertex buffer as shown in the following code fragment.

```
g_D3DDevice.DrawPrimitive D3DPT_TRIANGLELIST, 0, 1
```

The first parameter accepted by **DrawPrimitive** is a flag that tells Direct3D what type of primitives to draw. This sample uses the flag D3DPT_TRIANGLELIST to specify a list of triangles. The second parameter is the index of the first vertex to load. The third parameter tells the number of primitives to draw. Because this sample draws only one triangle, this value is set to 1.

For more information on different kinds of primitives, see **3-D Primitives**.

The last steps are to end the scene and then present the back buffer to the front buffer. This is shown in the following code fragment.

```
g_D3DDevice.EndScene
g_D3DDevice.Present ByVal 0, ByVal 0, 0, ByVal 0
```

After the back buffer is presented to the front buffer, the client window shows a triangle with three different colored points.

This tutorial has shown you how to use vertices to render geometric shapes. Tutorial 3: Using Matrices introduces the concept of matrices and how to use them.

Tutorial 3: Using Matrices

This tutorial introduces the concept of matrices and shows how to use them. The Vertices sample project rendered 2-D vertices to draw a triangle. However, in this tutorial you work with transformations of vertices in 3-D. Matrices are also used to set up cameras and viewports. For more information on working with transformation matrices, see **3-D Transformations**.

Before the Matrices sample project renders geometry, it calls the **SetupMatrices** application-defined function to create and set the matrix transformations that are used to render the 3-D triangle. Typically, three different types of transformation are applied to a 3-D scene. Steps for creating each one of these typical transformations are listed below.

- Step 1: Defining the World Transformation Matrix
- Step 2: Defining the View Transformation Matrix

- Step 3: Defining the Projection Transformation Matrix

For more information on matrices and transformations, see **Matrices**.

For more information on 3-D Transformations, see **3-D Transformations**.

Note

The path of the Matrices sample project is:
(SDK Root)
\Samples\Multimedia\VBSamples\Direct3D\Tutorials\Tut03_Matrices.
The order in which these transformation matrices are created does not affect the layout of the objects in a scene. However, Direct3D applies the matrices to the scene in the following order: (1) World ,(2) View, (3) Projection.
The code in the Matrices sample project is nearly identical to the code in Vertices sample project. This tutorial focuses only on the code unique to matrices and does not cover initializing Direct3D, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.
This tutorial also uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.

Step 1: Defining the World Transformation Matrix

The world transformation matrix defines how to translate, scale, and rotate geometry in 3-D model space.

The following code fragment rotates the triangle on the y-axis and then sets the current world transformation for the Microsoft® Direct3D® device.

```
Dim matWorld As D3DMATRIX
D3DXMatrixRotationY matWorld, Timer * 4
g_D3DDevice.SetTransform D3DTS_WORLD, matWorld
```

The first step is to rotate the triangle around the Y-axis by calling the **D3DXMatrixRotationY** method. The first parameter is a **D3DMATRIX** type that contains the returned matrix. The second parameter is the angle of rotation in radians.

The next step is to call **Direct3DDevice8.SetTransform** to set the current world transformation for the Direct3D device. The first parameter accepted by **SetTransform** tells which transformation to set. This sample uses the D3DTS_WORLD flag to specify that the world transformation should be set. The second parameter is a pointer to a matrix that is set as the current transformation.

For more information on world transformations, see **The World Transformation**.

After defining the world transformation for the scene, you can prepare the view transformation matrix. Again, note that the order in which transformations are defined is not critical. However, Direct3D applies the matrices to the scene in the following order: (1) World ,(2) View, (3) Projection.

Defining the view transformation matrix is described in Step 2: Defining the View Transformation Matrix.

Step 2: Defining the View Transformation Matrix

The view transformation matrix defines the position and rotation of the view. The view matrix is the camera for the scene.

The following code fragment creates the view transformation matrix and then sets the current view transformation to the Microsoft® Direct3D® device.

```
Dim matView As D3DMATRIX
D3DXMatrixLookAtLH matView, vec3(0#, 3#, -5#), _
                    vec3(0#, 0#,  0#), _
                    vec3(0#, 1#,  0#)

g_D3DDevice.SetTransform D3DTS_VIEW, matView
```

The first step is to define the view matrix by calling **D3DXMatrixLookAtLH**. The first parameter is a **D3DMATRIX** type that contains the returned matrix. The second, third, and fourth parameters define the eye point, look at point, and "up" direction. Here the eye is set back along the z-axis by five units and up three units, the look at point is set at the origin, and "up" is defined as the y-direction.

The next step is to call **Direct3DDevice8.SetTransform** to set the current view transformation for the Direct3D device. The first parameter accepted by **SetTransform** tells which transformation to set. This sample uses the D3DTS_VIEW flag to specify that the view transformation should be set. The second parameter is a pointer to a matrix that is set as the current transformation.

For more information on view transformations, see **The View Transformation**.

After defining the world transformation for the scene, you can prepare the projection transformation matrix. Again, note that the order in which transformations are defined is not critical. However, Direct3D applies the matrices to the scene in the following order: (1) World ,(2) View, (3) Projection.

Defining the projection transformation matrix is described in Step 3: Defining the Projection Transformation Matrix.

Step 3: Defining the Projection Transformation Matrix

The projection transformation matrix defines how geometry will be transformed from 3-D view space to 2-D viewport space.

The following code fragment creates the projection transformation matrix and then applies the transformation to the Microsoft® Direct3D® device.

```
Dim matProj As D3DMATRIX
D3DXMatrixPerspectiveFovLH matProj, g_pi / 4, 1, 1, 1000
g_D3DDevice.SetTransform D3DTS_PROJECTION, matProj
```

The first step is to call **D3DXMatrixPerspectiveFovLH** to set up the projection matrix. The first parameter is a **D3DMATRIX** type that contains the returned matrix. The second parameter defines the field of view, which tells how objects in the distance get smaller. A typical field of view is 1/4 pi, which is what the sample uses. The third parameter defines the aspect ratio. The sample uses the typical aspect ratio of 1. The fourth and fifth parameter define the near and far clipping plane. This determines the distance at which geometry should no longer be rendered. The Matrices sample project has its near clipping plane set at 1 and its far clipping plane set at 100.

The next step is to call **Direct3DDevice8.SetTransform** to set the current projection transformation for the Direct3D device. The first parameter accepted by **SetTransform** tells which transformation to set. This sample uses the D3DTS_PROJECTION flag to specify that the projection transformation should be set. The second parameter is a pointer to a matrix that is set as the current transformation.

For more information on projection transformations, see The Projection Transformation.

This tutorial has shown you how to use matrices. Tutorial 4: Creating and Using Lights shows how to add lights to your scene for more realism.

Tutorial 4: Creating and Using Lights

Microsoft® Direct3D® lights add more realism to 3-D objects. When used, each geometric object in the scene will be lit based on the location and type of lights that are used. The Lights sample project introduces the topics of lights and materials.

This tutorial takes the following step to create and use materials and lights.

- Step 1: Initializing Scene Geometry
- Step 2: Setting up Material and Light

Note

The path of the Lights sample project is:
(SDK Root)
\Samples\Multimedia\VBSamples\Direct3D\Tutorials\Tut04_Lights.
The code in the Lights sample project is nearly identical to the code in the Matrices sample project. This tutorial focuses only on the code unique to lights and does not cover initializing Direct3D, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.
This tutorial also uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.

This tutorial also makes use of matrices to transform geometry. For more information on matrices and transformations, see Tutorial 3: Using Matrices.

Step 1: Initializing Scene Geometry

One of the requirements of using lights is that each surface has a normal. To do this, the Lights sample project uses a different custom vertex type. The new custom vertex format has a 3-D position and a surface normal. The surface normal is used internally by Microsoft® Direct3D for lighting calculations.

```
' A structure for the custom vertex type.
Private Type CUSTOMVERTEX
    postion As D3DVECTOR    '3d position for vertex
    normal As D3DVECTOR     'surface normal for vertex
End Type

' The custom FVF, which describes the custom vertex structure.
Const D3DFVF_CUSTOMVERTEX = (D3DFVF_XYZ Or D3DFVF_NORMAL)
```

Now that the correct vector format is defined, the Lights sample project calls **InitGeometry**, which is an application-defined function that creates a cylinder. The first step is to fill the vertex buffer with the points of the cylinder. Note that in the following code fragment, each point is defined by a position and a normal.

```
For i = 0 To 49
    theta = (2 * g_pi * i) / (50 - 1)
    Vertices(2 * i + 0).postion = vec3(Sin(theta), -1, Cos(theta))
    Vertices(2 * i + 0).normal = vec3(Sin(theta), 0, Cos(theta))
    Vertices(2 * i + 1).postion = vec3(Sin(theta), 1, Cos(theta))
    Vertices(2 * i + 1).normal = vec3(Sin(theta), 0, Cos(theta))
Next
```

The second step is to create a vertex buffer that stores the points of the cylinder as shown in the following code fragment.

```
Set g_VB = g_D3DDevice.CreateVertexBuffer(VertexSizeInBytes * 50 * 2, _
            0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT)
If g_VB Is Nothing Then Exit Function
```

Now fill the vertex buffer with the vertices for the cylinder and the vertex buffer is ready for rendering. But first, the material and light for this scene must be set up before rendering the cylinder, as described in Step 2: Setting up Material and Light.

Step 2: Setting up Material and Light

To use lighting in Microsoft® Direct3D®, you must create one or more lights. To determine which color a geometric object reflects, a material is created that is used to

render geometric objects. Before rendering the scene, the Lights sample project calls an application-defined function named **SetupLights** to set up one material and one directional light. This function takes the following steps to create a material and a light.

- Step 2.1: Creating a Material
- Step 2.2: Creating a Light

Step 2.1: Creating a Material

A material defines the color that is reflected off the surface of a geometric object when a light hits it. The following code fragment uses the **D3DMATERIAL8** structure to create a material that is yellow.

```
Dim mtrl As D3DMATERIAL8
With col:    .r = 1: .g = 1: .b = 0: .a = 1:   End With
mtrl.diffuse = col
mtrl.Ambient = col
g_D3DDevice.SetMaterial mtrl
```

The diffuse color and ambient color for the material are set to yellow. The call to the **Direct3DDevice8.SetMaterial** method applies the material to the Microsoft® Direct3D® device used to render the scene. The only parameter that **SetMaterial** accepts is the address of the material to set. After this call is made, every primitive will be rendered with this material until another call is made to **SetMaterial** that specifies a different material.

Now that material has been applied to the scene, the next step is to create a light. This is described in Step 2.2: Creating a Light.

Step 2.2: Creating a Light

There are three types of lights available in Microsoft® Direct3D®. These are point lights, directional lights, and spotlights. The Lights sample project creates a directional light, which is a light that goes in one direction and oscillates the direction of the light.

The following code fragment uses the **D3DLIGHT8** structure to define a directional light.

```
Dim light As D3DLIGHT8
light.Type = D3DLIGHT_DIRECTIONAL
```

The following code fragment sets the diffuse color for this light to white.

```
light.diffuse.r = 1#
light.diffuse.g = 1#
light.diffuse.b = 1#
```

The following code fragment rotates the direction of the light around in a circle.

```
light.Direction.x = Cos(Timer * 2)
light.Direction.y = 1#
light.Direction.z = Sin(Timer * 2)
```

A range can be specified to tell Direct3D how far the light will have an effect. This member does not affect directional lights. The following code fragment assigns a range of 1000 units to this light.

```
light.Range = 1000#
```

The following code fragment assigns the light to the Direct3D device by calling **Direct3DDevice8.SetLight**.

```
g_D3DDevice.SetLight 0, light     'Let d3d know about the light.
```

The first parameter that **SetLight** accepts is the index that this light will be assigned to. Note that if a light already exists at that location, then it will be overwritten by the new light. The second parameter is a pointer to the light structure that defines the light. The Lights sample project places this light at index 0.

The code following fragment enables the light by calling **Direct3DDevice8.LightEnable**.

```
g_D3DDevice.LightEnable 0, 1
```

The first parameter that **LightEnable** accepts is the index of the light to enable. The second parameter is a value that tells whether to turn the light on (1) or off (0). In the Lights sample project, the light at index 0 is turned on.

The next step that this sample takes is to turn on ambient lighting by calling **Direct3DDevice8.SetRenderState**.

```
g_D3DDevice.SetRenderState D3DRS_AMBIENT, &H202020
```

The two parameters that **SetRenderState** accepts is which device state variable to modify and what value to set it to. The sample sets the D3DRS_AMBIENT device variable to a light gray color (&H202020). Ambient lighting will light up all objects by the given color.

Before using lights in Direct3D, they must be enabled. The following code fragment tells Direct3D to render lights by calling **SetRenderState**.

```
g_D3DDevice.SetRenderState D3DRS_LIGHTING, 1    'Make sure lighting is enabled.
```

This sample sets the D3DRS_LIGHTING device variable to TRUE, which has the effect of enabling the rendering of lights.

For more information on lighting, see **Mathematics of Direct3D Lighting**.

This tutorial has shown you how to use lights and materials. Tutorial 5: Using Texture Maps shows you how to add texture to surfaces.

Tutorial 5: Using Texture Maps

While lights and materials add a great deal of realism to a scene, nothing adds more realism than adding textures to surfaces. Textures can be thought of as wallpaper that is shrink-wrapped onto a surface. You could put a wood texture on a cube to make it look like the cube is actually made of wood. The Texture sample project adds a banana peel texture to the cylinder created in the Lights sample. This tutorial covers how to load textures, set up vertices, and display objects with texture.

This tutorial implements textures with the following steps:

- Step 1: Defining a Custom Vector Format
- Step 2: Initializing Screen Geometry
- Step 3: Rendering the Scene

Note

The path of the Textures sample project is:
(SDK Root)
\Samples\Multimedia\VBSamples\Direct3D\Tutorials\Tut05_Textures.
The code in the Texture sample project is nearly identical to the code in the Lights sample project, except that the Texture sample project does not create a material or a light. This tutorial focuses only on the code unique to textures and does not cover initializing Microsoft® Direct3D®, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.
This tutorial also uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.
This tutorial also makes use of matrices to transform geometry. For more information on matrices and transformations, see Tutorial 3: Using Matrices.

Step 1: Defining a Custom Vector Format

Before using textures, a custom vertex format that includes texture coordinates must be used. Texture coordinates tells Microsoft® Direct3D® where to place a texture for each vector in a primitive. Texture coordinates range from 0.0 to 1.0, where (0.0, 0.0) represents the top left-hand side of the texture and (1.0, 1.0) represents the bottom right-hand side of the texture.

The following code fragment shows how the Texture sample project sets up it's custom vertex format to include texture coordinates.

```
' Custom vertex type.
Private Type CUSTOMVERTEX
    postion As D3DVECTOR    '3-D position for vertex.
    color As Long           'Color of the vertex.
    tu As Single            'Texture map coordinate.
    tv As Single            'Texture map coordinate.
End Type
```

```
' Custom FVF.
Const D3DFVF_CUSTOMVERTEX = (D3DFVF_XYZ Or D3DFVF_DIFFUSE Or D3DFVF_TEX1)
```

For more information on texture coordinates, see Texture Coordinates.

Now that a custom vertex type has been defined, the next step is to load a texture and create a cylinder, as described in Step 2: Initializing Screen Geometry.

Step 2: Initializing Screen Geometry

Before rendering, the Texture sample project calls an application-defined function, **InitGeometry**, to create a texture and set up the geometry for a cylinder.

Textures are created from file-based images. The following code fragment uses **D3DX8.CreateTextureFromFile** to create a texture from Banana.bmp that will be used to cover the surface of the cylinder.

```
Set g_Texture = g_D3DX.CreateTextureFromFile(g_D3DDevice, App.Path + "\banana.bmp")
If g_Texture Is Nothing Then Exit Function
```

The first parameter that **CreateTextureFromFile** accepts is the Microsoft® Direct3D® device that will be used to render the texture. The second parameter is a string that specifies the filename from which to create the texture. This sample specifies Banana.bmp to load the image from that file.

The banana texture is loaded and ready to use. The next step is to create the cylinder. The following code fills the vertex buffer with a cylinder. Note that each point has the texture coordinate (tu, tv).

```
For i = 0 To 49
    theta = (2 * g_pi * i) / (50 - 1)

    Vertices(2 * i + 0).position = vec3(Sin(theta), -1, Cos(theta))
    Vertices(2 * i + 0).color = &HFFFFFFFF  'White.
    Vertices(2 * i + 0).tu = i / (50 - 1)
    Vertices(2 * i + 0).tv = 1

    Vertices(2 * i + 1).position = vec3(Sin(theta), 1, Cos(theta))
    Vertices(2 * i + 1).color = &HFF808080  'Grey.
    Vertices(2 * i + 1).tu = i / (50 - 1)
    Vertices(2 * i + 1).tv = 0
Next
```

Each vertex includes a position, color, and texture coordinates. The Texture sample project sets the texture coordinates for each point so that the texture will wrap smoothly around the cylinder.

Now that the texture is loaded and the vertex buffer is ready for rendering, it is time to render the display, as described in Step 3: Rendering the Scene.

Step 3: Rendering the Scene

After geometry has been initialized, it is time to render the scene. In order to render an object with texture, the texture must be set as one of the current textures. The next step is to set the texture stage states values. Texture stage states enable you to define the behavior of how a texture or textures are to be rendered. For example, you could blend multiple textures together.

The Texture sample project starts by setting the texture to use. The following code fragment sets the texture that the Microsoft® Direct3D® device will use to render with **Direct3DDevice8.SetTexture**.

```
g_D3DDevice.SetTexture 0, g_Texture
```

The first parameter that **SetTexture** accepts is a stage identifier to set the texture to. A device can have up to eight set textures, so the maximum value here is 7. This sample only has one texture and places it at stage 0. The second parameter is a pointer to a texture object. This sample uses the texture object that it created in its **InitGeometry** application-defined function.

The following code fragment sets the texture stage state values by calling the **Direct3DDevice8.SetTextureStageState** method.

```
g_D3DDevice.SetTextureStageState 0, D3DTSS_COLOROP, D3DTOP_MODULATE
g_D3DDevice.SetTextureStageState 0, D3DTSS_COLORARG1, D3DTA_TEXTURE
g_D3DDevice.SetTextureStageState 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE
g_D3DDevice.SetTextureStageState 0, D3DTSS_ALPHAOP, D3DTOP_DISABLE
```

The first parameter that **SetTextureState** accepts is the stage index for the state variable to be set. This sample is changing the values for the texture at stage 0, so it puts a 0 here. The next parameter is the texture state to set. For a list of all valid texture states and their meaning, see **CONST_D3DTEXTURESTAGESTATETYPE**. The next parameter is the value to set the texture state to. The value that you place here is based on the texture stage state value that you are modifying.

After setting up the desired values for each texture stage state, the cylinder can be rendered with texture added to the surface.

Another way to use texture coordinates is to have them automatically generated. This is done by using a texture coordinate index (TCI). The TCI uses a texture matrix to transform the (x,y,z) TCI coordinates into (tu, tv) texture coordinates. In the Texture sample project, the position of the vertex in camera space is used to generate texture coordinates.

The first step is to create the matrix that will be used for the transformation. The code fragment for creating the matrix is shown below.

```
Dim mat As D3DMATRIX

mat.m11 = 0.25: mat.m12 = 0#:     mat.m13 = 0#: mat.m14 = 0#
```

```
mat.m21 = 0#:   mat.m22 = -0.25:  mat.m23 = 0#: mat.m24 = 0#
mat.m31 = 0#:   mat.m32 = 0#:     mat.m33 = 1#: mat.m34 = 0#
mat.m41 = 0.5:  mat.m42 = 0.5:    mat.m43 = 0#: mat.m44 = 1#
```

After the matrix is created, it must be set by calling **SetTransform**, as shown in the following code segment.

```
g_D3DDevice.SetTransform D3DTS_TEXTURE0, mat
```

The D3DTS_TEXTURE0 flag tells Direct3D to apply the transformation to the texture located at texture stage 0. The next step that this sample takes is to set more texture stage state values to get the desired effect. That is done in the code fragment below.

```
g_D3DDevice.SetTextureStageState 0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTFF_COUNT2
g_D3DDevice.SetTextureStageState 0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACEPOSITION
```

The texture coordinates are set up and now the scene is ready to be rendered. Notice that the coordinates are automatically created for the cylinder. This particular setup gives the effect of the texture being laid over the rendering screen after the geometric shapes have been rendered.

For more information on textures, see Textures.

This tutorial has shown you how to use textures for surfaces. Tutorial 6: Using Meshes shows you how to use complex geometric shapes with meshes.

Tutorial 6: Using Meshes

Complicated geometry is usually modeled using 3-D modeling software and saved to a file. A example of this is the .x file format. Microsoft® Direct3D® uses meshes to load the objects from these files. Meshes are somewhat involved, but D3DX contains functions to help out. The Meshes sample project introduces the topic of meshes and shows how to load, render, and unload a mesh.

This tutorial shows how to load and render a mesh with the following steps.

- Step 1: Loading a Mesh Object
- Step 2: Rendering a Mesh Object

The path of the Meshes sample project is:
(SDK Root)
\Samples\Multimedia\VBSamples\Direct3D\Tutorials\Tut06_Meshes.
The code in the Meshes sample project is nearly identical to the code in the Lights sample project, except that the Meshes sample project does not create a material or a light. This tutorial focuses only on the code unique to Meshes and does not cover

initializing Direct3D, rendering, or shutting down. For information on these tasks, see Tutorial 1: Creating a Device.
This tutorial also uses custom vertices and a vertex buffer to display geometry. For more information on selecting a custom vertex type and implementing a vertex buffer, see Tutorial 2: Rendering Vertices.
This tutorial also makes use of matrices to transform geometry. For more information on matrices and transformations, see Tutorial 3: Using Matrices.
This tutorial also uses textures to cover the surface of the mesh. For more information on loading and using textures, see Tutorial 5: Using Texture Maps.

Step 1: Loading a Mesh Object

A Microsoft® Direct3D® application must first load a mesh before using it. The Meshes sample project loads the tiger mesh by calling **InitGeometry**, an application-defined function, after loading the required Direct3D objects.

A mesh needs a material buffer that will store information about all the materials and textures that will be used. The function starts by declaring a material buffer.

```
Dim MtrlBuffer As D3DXBuffer
```

The Meshes sample project uses the **D3DX8.LoadMeshFromX** method to load the mesh, as shown in the code below.

```
Set g_Mesh = g_D3DX.LoadMeshFromX(App.Path + "\Tiger.x", D3DXMESH_MANAGED, _
                    g_D3DDevice, Nothing, MtrlBuffer, g_NumMaterials)

If g_Mesh Is Nothing Then Exit Function
```

The first parameter that **D3DXLoadMeshFromX** accepts is a string that tells the name of the Microsoft DirectX® file to load. This sample loads the tiger mesh from Tiger.x. The second parameter tells Direct3D how to create the mesh. The sample uses the D3DXMESH_MANAGED flag, which tells Direct3D to use the D3DPOOL_MANAGED memory class for the index buffers. The third parameter is the Direct3D device that will be used to render the mesh. The fourth parameter takes a **D3DXBuffer** object. This object will be filled with information about neighbors for each face. This information is not required for this sample, so this parameter is set to Nothing. The fifth parameter also takes a pointer to a **D3DXBuffer** object. After this method is finished, this object will be filled with D3DXMATERIAL structures for the mesh. The sixth parameter takes a variable of type Long that will tell the number of D3DXMATERIAL structures placed into the *RetMaterials* parameter after this method returns.

After loading the mesh object and material information, you need to extract the material properties and texture names from the material buffer.

Before getting this information from the material buffer, the material and texture arrays must be resized to fit all the materials and textures for this mesh.

```
ReDim g_MeshMaterials(g_NumMaterials)
ReDim g_MeshTextures(g_NumMaterials)
```

For each material in the mesh the following steps occur.

The first step is to use **D3DX8.BufferGetMaterial** to copy the material, as shown in the code fragment below.

```
g_D3DX.BufferGetMaterial MtrlBuffer, i, g_MeshMaterials(i)
```

The first parameter that **BufferGetMaterial** accepts is the material object, which in this case is MtrlBuffer. The second parameter is the index of the material to retrieve. The third parameter is the location to place the material.

The second step is to set the ambient color for the material, as shown in the following code fragment.

```
g_MeshMaterials(i).Ambient = g_MeshMaterials(i).diffuse
```

The final step is to create the texture for the material, as shown in the following code fragment.

```
strTexName = g_D3DX.BufferGetTextureName(MtrlBuffer, i)
If strTexName <> "" Then
   Set g_MeshTextures(i) = g_D3DX.CreateTextureFromFile(g_D3DDevice, App.Path + "\" +
strTexName)
End If
```

After loading all the materials, you need to let Direct3D know that you are finished with the material buffer by setting it to Nothing.

```
Set MtrlBuffer = Nothing
```

The mesh, along with the corresponding materials and textures are loaded. The mesh is ready to be rendered to the display, as described in Step 2: Rendering a Mesh Object.

Step 2: Rendering a Mesh Object

The mesh is loaded and ready to be rendered. The mesh is divided into a subsets for each material that was loaded for the mesh. To render each subset, the mesh is rendered in a loop. The first step in the loop is to set the material for the subset, as shown in the following code fragment.

```
g_D3DDevice.SetMaterial g_MeshMaterials(i)
```

The second step in the loop is to set the texture for the subset, as shown in the following code fragment.

```
g_D3DDevice.SetTexture 0, g_MeshTextures(i)
```

After setting the material and texture, the subset is drawn with the **D3DXBaseMesh.DrawSubset** method, as shown in the following code fragment.

```
g_Mesh.DrawSubset i
```

The **DrawSubset** method takes a **Long** value that specifies which subset of the mesh to draw. This sample uses the value that is incremented each time the loop runs.

This loop will run until each subset in the mesh has been rendered.

This tutorial has shown you how to load and render meshes. This is the last tutorial in this section. To find out how a typical Microsoft® Direct3D® application is written in Microsoft Visual Basic®, check the DirectX Graphics Visual Basic Samples.

DirectX Graphics C/C++ Samples

The following sample applications demonstrate the use and capabilities of the Microsoft® Direct3D® and Direct3DX application programming interfaces.

- Billboard Sample
- BumpEarth Sample
- BumpLens Sample
- BumpUnderwater Sample
- BumpWaves Sample
- DotProduct3 Sample
- Emboss Sample
- ClipMirror Sample
- DolphinVS Sample
- DXTex Tool
- EnhancedMesh Sample
- CubeMap Sample
- FishEye Sample
- SphereMap Sample
- MFCFog Sample
- MFCPixelShader Sample
- MFCTex Sample
- OptimizedMesh Sample
- Pick Sample
- PointSprites Sample

- ProgressiveMesh Sample
- RTPatch Sample
- SkinnedMesh Sample
- ShadowVolume Sample
- StencilDepth Sample
- StencilMirror Sample
- Text3D Sample
- VertexBlend Sample
- VertexShader Sample
- VolumeTexture Sample
- Water Sample

Although Microsoft DirectX® samples include Microsoft Visual C++® project workspace files, you might need to verify other settings in your development environment to ensure that the samples compile properly. For more information, see Compiling DirectX Samples and Other DirectX Applications.

See Also

DirectX Graphics C/C++ Tutorials

Billboard Sample

Description

The Billboard sample illustrates the billboarding technique. Rather than rendering complex 3-D models, such as a high-polygon tree model, billboarding renders a 2-D image of the model and rotates it to always face the eyepoint. This technique is commonly used to render trees, clouds, smoke, explosions, and more. For more information, see Billboarding.

In this sample, a camera flies around a 3-D scene with a tree-covered hill. The trees look like 3-D objects, but they are actually 2-D billboarded images that are rotated towards the eyepoint. The hilly terrain and the skybox, a six-sided cube containing sky textures, are objects loaded from .x files. They are used for visual effect and are unrelated to the billboarding technique.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\Billboard

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The billboarding technique is the focus of this sample. The camera is moved in each frame, so the viewpoint changes accordingly. As the viewpoint changes, a rotation matrix is generated to rotate the billboards about the y-axis so that they face the new viewpoint. The computation of the billboard matrix occurs in the **FrameMove** function. The trees are also sorted in this function, as required for proper alpha blending, because billboards typically have some transparent pixels. The trees are rendered from a vertex buffer in the **DrawTrees** function.

The billboards in this sample are constrained to rotate about the y-axis only. Otherwise, the tree trunks would appear to not be fixed to the ground. For effects such as explosions in a 3-D flight simulator or space shooter, billboards are typically not constrained to one axis.

This sample uses common Microsoft® DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the source code in (SDK Root) \Samples\Multimedia\VBSamples.

BumpEarth Sample

Description

The BumpEarth sample demonstrates the bump mapping capabilities of Microsoft® Direct3D®. Bump mapping is a texture-blending technique that renders the appearance of rough, bumpy surfaces. This sample renders a rotating, bump-mapped planet Earth.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, see Bump Mapping.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\BumpMapping\BumpEarth

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

BumpLens Sample

Description

The BumpLens sample demonstrates a lens effect that can be achieved using bump mapping. Bump mapping is a multitexture-blending technique that renders the appearance of rough, bumpy surfaces, but can also be used for other effects, as shown here.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, see Bump Mapping.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\BumpMapping\BumpLens

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

This sample uses bump mapping in a nontraditional fashion. Because bump mapping only perturbs an environment map, it can be used for other effects. In this case, it is used to perturb a background image, which can be rendered on the fly, to make a lens effect.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in the following directory: (SDK Root) \Samples\Multimedia\Common

BumpUnderwater Sample

Description

The BumpUnderwater sample demonstrates an underwater effect that can be achieved using bump mapping. Bump mapping is a multitexture-blending technique that renders the appearance of rough, bumpy surfaces, but can also be used for other effects as shown here.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, see Bump Mapping.

Path

Source: (SDK Root)
\Samples\Multimedia\Direct3D\BumpMapping\BumpUnderWater

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

This sample uses bump mapping in a nontraditional fashion. Because bump mapping only perturbs an environment map, it can be used for other effects. In this case, it is used to perturb a background image, which can be rendered on the fly, to make an underwater effect.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root)
\Samples\Multimedia\Common.

BumpWaves Sample

Description

The BumpWaves sample demonstrates the bump mapping capabilities of Microsoft® Direct3D®. Bump mapping is a multitexture-blending technique that renders the appearance of rough, bumpy surfaces. This sample renders a waterfront scene with only four triangles. The waves in the scene are completely fabricated with a bump map.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, refer to the Microsoft DirectX® SDK documentation.

This sample also uses a technique called projected textures, which is a texture-coordinate generation technique and is not the focal point of the sample. For more information on texture-coordinate generation, see Bump Mapping.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\BumpMapping\BumpWaves

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

In this sample, bump mapping is used to generate waves in a scene. The backdrop is used as a projective texture for the environment map, so it reflects in the waves. The

waves themselves appear to be generated with many polygons. However, it is one large quad.

This sample uses common Microsoft DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

DotProduct3 Sample

Description

The DotProduct3 sample demonstrates an alternative approach to Microsoft® Direct3D® bump mapping. This technique is named after the mathematical operation that combines a light vector with a surface normal. The normals for a surface are traditional (x,y,z) vectors stored in RGBA format in a texture map—called a normal map for this technique.

Not all cards support DotProduct3 blending texture stages, and not all cards support Direct3D bump mapping.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\BumpMapping\DotProduct3

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The lighting equation for simulating bump mapping involves using the dot product of the surface normal and the lighting vector. The lighting vector is passed into the texture factor, and the normals are encoded in a texture map. The blend stages look like this:

```
SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_DOTPRODUCT3 );
```

```
SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TFACTOR );
```

The next step is to store the normals in the texture. To do this, the components of a vector (XYZW) are each turned from a 32-bit floating value into a signed 8-bit integer and packed into a texture color (RGBA). The code shows how to do this using a custom-generated normal map, as well as one built from a bump mapping texture image.

Not all cards support all features for bump mapping techniques. Some hardware has no, or limited, bump mapping support. For more information on bump mapping, refer to the Microsoft® DirectX® SDK documentation.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

Emboss Sample

Description

The Emboss sample demonstrates an alternative approach to Microsoft® Direct3D® bump mapping. Embossing is done by subtracting the height map from itself and having texture coordinates that are slightly changed.

Not all cards support Direct3D bump mapping. Refer to the Microsoft DirectX® SDK documentation for more information.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\Emboss

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
CTRL-E	Turns emboss effect on/off.

Programming Notes

A bump map is a texture that stores the perturbation data. Bump mapping requires two textures. One is an environment map, which contains the lights that you see in the scene. The other is the actual bump mapping, which contain values—stored as du and dv—used to bump the environment map's texture coordinates. Some bump maps also contain luminance values to control the shininess of a particular texel.

In this sample, bump mapping is used to emboss the surface of the tiger.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

ClipMirror Sample

Description

The ClipMirror sample demonstrates the use of custom-defined clip planes. A 3-D scene is rendered normally, and then in a second pass as if reflected in a planar mirror. Clip planes are used to clip the reflected scene to the edges of the mirror.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\ClipMirror

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

The mouse is also used in this sample to control the viewing position.

Programming Notes

The main feature of this sample is the use of clip planes. The rectangular mirror has four edges, so four clip planes are used. Each plane is defined by the eyepoint and two vertices of one edge of the mirror. With the clip planes in place, the view matrix is reflected in the mirror's plane, and then the scene geometry (the teapot object) can be rendered as normal. Afterward, a semi-transparent rectangle is drawn to represent the mirror itself.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

DolphinVS Sample

Description

The DolphinVS sample shows an underwater scene of a dolphin swimming, with caustic effects on the dolphin and sea floor. The dolphin is animated using a technique called tweening. The underwater effect simply uses fog, and the water caustics use an animated set of textures. These effects are achieved using vertex shaders.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\DolphinVS

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Several things are happening in this sample.

- Two vertex shaders are used: one for the dolphin and one for the sea floor. The vertex shaders are assembly instructions which are assembled from two files, DolphinTween.vsh and SeaFloor.vsh.
- The dolphin is tweened—a form of morphing—by passing three versions of the dolphin down in multiple vertex streams. The vertex shader takes the weighted positions of each mesh and produces an output position. The vertex is then lit, texture coordinates are computed, and fog is added.
- The sea floor is handled similarly, just with no need for tweening. The caustic effects are added in a separate alpha-blended pass, using an animated set of 32 textures. The texture coordinates for the caustics are generated in the vertex shaders, stored as TEXCOORDINDEX stage 1.

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

DXTex Tool

Description

The DXTex tool enables DirectX SDK users to create texture maps that use the new DXTn compression formats. Creating a DXTn-compressed texture is not difficult. However, Direct3D can do the conversion for you when using the **IDirect3DTexture8** interface Advanced developers will probably want to write their own tools that meet their specific needs, but the DXTex tool provides useful basic functionality.

Functionality

- Opens .bmp and DirectDrawSurface (DDS) files. For more information, see DDS File Format, below.
- Opens .bmp files as alpha channel, either explicitly or implicitly, using a <*filename*>_a.bmp naming convention.
- Saves textures in DDS format.
- Supports conversion to all five DXTn compression formats.
- Supports generation of mipmaps, using a box filter.
- Supports visualization of alpha channel as a grayscale image or through a user-selectable background color.
- Supports easy visual comparison of image quality between formats.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\DXTex

Executable: (SDK Root)\bin\dxutils

User Interface

DxTex uses a fairly traditional user interface (UI) in that each texture map is a document, and several documents can be open at a time. However, each document can hold the texture in either one or two formats at once while the document is open in DxTex. For example, you can import a .bmp file, which automatically creates a 32-bit ARGB texture. You can then choose to convert the texture to DXT1 format. The document now has the image open in both formats, and you can switch between the formats by clicking the window or. You can also switch by clicking **Original View** or **New View** on the **View** menu. This makes it easy for the you to observe any artifacts introduced by image compression, and to try different compression formats without progressive degradation of the image. For example, if this technique is not used and you convert an image from ARGB to DXT1, all but one bit of alpha is lost.

If the you then decide to convert to DXT2, there is still only two alpha levels. Using the DxTex system, the second conversion is done from the original ARGB format, and the DXT2 image will contain all 16 levels of alpha supported by DXT2. When the image is saved, the original format is discarded, and only the new format is stored.

Keep in mind the following when using the DxTex interface.

- If no format conversion has been requested since the document was opened, the one format is the original format—that is, the format in which the image will be written when the document is saved.
- If format conversion has been requested since the document was opened, the converted format is the new format—that is, the format in which the image will be written when the document is saved.
- If a second format conversion is requested, the new format from the first conversion is replaced with the second format. The original format will be unchanged.
- Generating mipmaps applies to both formats simultaneously.
- If the original format has multiple mipmaps, and conversion to a new format is requested, the new format will automatically have multiple mipmaps as well.

Performance

DxTex uses the Microsoft Direct3D® Reference Rasterizer to draw the textures, regardless of what 3-D hardware is available. So with larger textures (greater than 256x256 pixels), the application may be somewhat slow, depending on your CPU speed.

DSS File Format

The DxTex native file format is called the DDS file format because it encapsulates the information in a DirectDrawSurface. It has the following format:

```
DWORD dwMagic (0x20534444, or "DDS")
DDSURFACEDESC2 ddsd (provides information about the surface format)
BYTE bData1[] (surface data for the main surface)
[BYTE bData2[]...] (surface data for attached surfaces follows)
```

This format is easy to read and write to, and it supports features such as alpha and multiple mip levels, as well as DXTn compression. The Compress sample application in the DirectX SDK demonstrates how to properly import a DDS file.

It is not necessary to use the DDS format in order to use DXTn-compressed textures, or vice-versa, but the two work well together.

Mipmaps

Mipmapping is a technique that improves image quality and reduces texture memory bandwidth by providing prefiltered versions of the texture image at multiple resolutions.

To generate mipmaps in DxTex, the width and height of the source image must both be powers of 2. To do this, go to the **Format** menu, click **Generate Mip Maps**. Filtering is done through a simple box filter. That is, the four nearest pixels are averaged to produce each destination pixel.

Alpha

Many texture formats include an alpha channel, which provides opacity information at each pixel. DxTex fully supports alpha in textures. When you import a .bmp file, if there is a file of the same size with a name that ends in _a—for example, Sample.bmp and Sample_a.bmp—the second file is loaded as an alpha channel. The blue channel of the second .bmp is stored in the alpha channel. Once a document is open, you can explicitly load a .bmp file as the alpha channel by clicking **Open As Alpha Channel** on the **File** menu.

To view the alpha channel directly without the RGB channels, on the **View** menu, click **Alpha Channel Only**. The alpha channel appears as a grayscale image. If no alpha channel has been loaded, every pixel has a full alpha channel and the image appears solid white when viewing "Alpha Channel Only". To turn off alpha channel viewing, click the **Alpha Channel Only** command a second time.

In the usual view, the effect of the alpha channel is visible because the window has a solid background color that shows through the texture where the alpha channel is less than 100 percent. You can change this background color by clicking **Change Background Color** on the **View** menu. This choice does not affect the texture itself or the data that is written when the file is saved.

The DXT2 and DXT4 formats use premultiplied alpha. This means that the red, green, and blue values stored in the surface are already multiplied by the corresponding alpha value. Direct3D cannot copy from a surface that contains premultiplied alpha to one that contains non-premultiplied alpha, so some DxTex operations (Open as Alpha Channel, conversion to DXT3, and conversion to DXT5) are not possible on DXT2 and DXT4 formats. Supporting textures using these formats

is difficult on Direct3D devices that do not support DXTn textures. This is because Direct3D cannot copy them to a traditional ARGB surface either, unless that ARGB surface uses premultiplied alpha as well, which is rare. For this reason, you might find it easier to use DXT3 rather than DXT2, and DXT5 rather than DXT4 when possible.

Command Line Options

You can use command-line options to pass input files, an output file name, and processing options to DxTex. If an output file name is specified, the application exits automatically after writing the output file, and no user interface is presented.

```
dxtex [infilename] [-a alphaname] [-m] [DXT1|DXT2|DXT3|DXT4|DXT5] [outfilename]

infilename:          Name of the file to load.  This can be a
                     .bmp or DDS file.

-a alphaname:        The next parameter is the name of a .bmp
                     file to load as the alpha channel. If no
                     alpha filename is specified, DxTex still
                     looks for a file named Infilename_a.bmp. If
                     it exists, use that file as the alpha
                     channel.

-m:                  Mipmaps are generated.

DXT1|DXT2|DXT3|DXT4|DXT5: Compression format. If no format is
                     specified, the image will be in ARGB-8888.

outfilename:         Name of the destination file.  If this
                     option is not specified, the user interface
                     shows the current file and all requested
                     operations.  If an outfilename is specified,
                     the application exits after saving the
                     processed file without presenting a user
                     interface.
```

EnhancedMesh Sample

Description

The EnhancedMesh sample shows how to use D3DX to load and enhance a mesh. The mesh is enhanced by increasing the vertex count.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\EnhancedMesh

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

CubeMap Sample

Description

The CubeMap sample demonstrates an environment mapping technique called cube mapping. Environment mapping is a technique in which the environment surrounding a 3-D object, such as the lights, is put into a texture map, so that the object can have complex lighting effects without expensive lighting calculations.

Not all cards support all features for environment mapping techniques, such as cube mapping and projected textures. For more information on environment mapping, cube mapping, and projected textures, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\EnvMapping\CubeMap

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Cube mapping is a technique that employs a six-sided texture. To visualize the effects of this technique, imagine a wall-papered room in which the wallpaper has been shrink-wrapped around an object. Cube mapping is superior to sphere-mapping because the latter is inherently view-dependent; sphere maps are constructed for one particular viewpoint in mind. Cube maps also have no geometry distortions, so they can be generated on the fly using **IDirect3DDevice8::SetRenderTarget** for all six cube map faces.

Cube mapping works with Microsoft Direct3D® texture coordinate generation. By setting D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR, Direct3D generates cube map texture coordinates from the reflection vector for a vertex, thereby making this technique easy for environment mapping effects where the environment is reflected in the object.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

FishEye Sample

Description

The FishEye sample shows a fish-eye lens effect that can be achieved using cube maps.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\EnvMapping\FishEye

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The FishEye sample scene is rendered into the surfaces of a cube map, rather than the back buffer. Then the cube map is used as an environment map to reflect the scene in some distorted geometry—in this case, some geometry approximating a fish eye lens. See the CubeMap sample for more information about cube mapping.

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

SphereMap Sample

Description

The SphereMap sample demonstrates an environment mapping technique called sphere mapping. Environment mapping is a technique in which the environment surrounding a 3-D object, such as the lights, are put into a texture map, so that the object can have complex lighting effects without expensive lighting calculations.

Not all cards support all features for environment mapping techniques, such as cube mapping and projected textures. For more information on environment mapping, cube mapping, and projected textures, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\EnvMapping\SphereMap

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.

SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Sphere-mapping uses a precomputed (at model time) texture map that contains the entire environment as reflected by a chrome sphere. The idea is to consider each vertex, compute its normal, find where the normal matches on the chrome sphere, and then assign that texture coordinate to the vertex.

Although the math is not complicated, this involves computations for each vertex for every frame. Direct3D has a texture-coordinate generation feature that you can use to do this. The relevant render state operation is D3DTSS_TCI_CAMERASPACENORMAL, which takes the normal of the vertex in camera space and puts it through a texture transform to generate texture coordinates. Then you set up the texture matrix to do the rest. In this simple case, the matrix only has to scale and translate the texture coordinates to get from camera space (-1, +1) to texture space (0,1).

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

MFCFog Sample

Description

The MFCFog sample illustrates how to use Microsoft® Direct3D® with Microsoft Foundation Classes (MFC), using a CFormView. Various controls are used to control fog parameters for the 3-D scene.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\MFCFog

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

All user interaction for this sample is implemented through the visible MFC controls —sliders, radio buttons, and so on. You are encouraged to play with the controls and observe the various effects they have on the rendered 3-D scene.

Programming Notes

All the MFC code is contained with the CFormView class's derived member functions. You can find the MFC code and D3D initialization code in the D3dapp.cpp source file. This file can be ported to work with another application by stripping out the fog-related code.

The Direct3D fog code is contained in Fog.cpp. It includes functions to initialize, animate, render, and clean up the scene.

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

MFCPixelShader Sample

Description

The MFCFog sample illustrates how to use Microsoft® Direct3D® with Microsoft Foundation Classes (MFC), using a CFormView. Various controls are used to the control the pixel shader used for the 3-D scene.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\MFCPixelShader

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

All user interaction for this sample is implemented through the visible MFC controls —sliders, radio buttons, and so on. You are encouraged to play with controls and observe the various effects they have on the rendered 3-D scene.

Programming Notes

All the MFC code is contained with the CFormView class's derived member functions. You can find the MFC code and D3D initialization code in the D3dapp.cpp source file. This file can be ported to work with another application by stripping out the fog-related code.

The Direct3D fog code is contained in Fog.cpp. It includes functions to initialize, animate, render, and clean up the scene.

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

MFCTex Sample

Description

The MFCTex sample illustrates how to use Microsoft® Direct3D® with Microsoft Foundation Classes (MFC), using a CFormView. Various controls are used to the control the texture appearance.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\MFCTex

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

All user interaction for this sample is implemented through the visible MFC controls —sliders, radio buttons, and so on. You are encouraged to play with controls and observe the various effects they have on the rendered 3-D scene.

Programming Notes

All the MFC code is contained with the CFormView class's derived member functions. You can find the MFC code and D3D initialization code in the D3dapp.cpp source file. This file can be ported to work with another application by stripping out the fog-related code.

The Direct3D fog code is contained in Fog.cpp. It includes functions to initialize, animate, render, and clean up the scene.

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

OptimizedMesh Sample

Description

The OptimizedMesh sample illustrates how to load and optimize a file-based mesh using the D3DX mesh utility functions.

For more information on D3DX, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\OptimizedMesh

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
CTRL-O	Opens mesh file.
CTRL-M	Toggles optimized mesh.

Programming Notes

Many Microsoft Direct3D® samples in the DirectX SDK use file-based meshes. However, the OptimizedMesh sample is a good example of the basic code necessary for loading a mesh. The D3DX mesh loading functionality collapses the frame hierarchy of an .x file into one mesh.

For other samples, the bare bones D3DX mesh functionality is wrapped in a common class CD3DMesh. If you want to keep the frame hierarchy, you can use the common class CD3DFile.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

Pick Sample

Description

The Pick sample shows how to implement picking; that is, finding which triangle in a mesh is intersected by a ray. In this case, the ray comes from mouse coordinates.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\Pick

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Use the mouse to pick any spot in the mesh to see that triangle.

Programming Notes

When you click the mouse, the code reads the screen coordinates of the cursor. These coordinates are converted, through the projection and view matrices, into a ray that goes from the eyepoint through the point clicked on the screen and into the scene. This ray is passed to IntersectTriangle along with each triangle of the loaded model to determine which triangles, if any, are intersected by the ray. The texture coordinates of the intersected triangle is also determined.

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

PointSprites Sample

Description

The PointSprites sample shows how to use the new Microsoft® Direct3D® point sprites feature. A point sprite is simply a forward-facing, textured quad that is referenced only by (x,y,z) position coordinates. Point sprites are most often used for particle systems and related effects.

Not all cards support all features for point sprites. For more information on point sprites, refer to the Microsoft DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\PointSprites

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
W, S	Moves forward and back.
E, Q	Turns left and right.
A, Z	Rotates up and down.
ARROW KEYS	Slides left, right, up, and down.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.
F3	Animates the emitter.
F4	Changes color.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Without Direct3D support, point sprites can be implemented with four vertices that are oriented each frame towards the eyepoint, much like a billboard. With Direct3D, though, you can refer to each point sprite just by its center position and a radius. This saves processor computation time and bandwidth when uploading vertex information to the graphics card.

In this sample, each particle is implemented using multiple alpha-blended point sprites, giving the particle a motion-blur effect.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

ProgressiveMesh Sample

Description

The ProgressiveMesh sample illustrates how to load and optimize a file-based mesh using the D3DX mesh utility functions. A progressive mesh is one in which the vertex information is stored internally in a special tree that can be accessed to render the mesh with any number of vertices. This procedure is fast, so progressive meshes are

ideal for level-of-detail scenarios, where objects in the distance are rendered with fewer polygons.

For more information on D3DX, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\ProgressiveMesh

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
UP ARROW	Adds one vertex to the progressive mesh.
DOWN ARROW	Subtracts one vertex from the progressive mesh.
PAGE UP	Adds 100 vertices to the progressive mesh.
PAGE DOWN	Subtracts 100 vertices from the progressive mesh.
HOME	Displays all available vertices for the progressive mesh.
END	Displays the minimum vertices for the progressive mesh.

Programming Notes

Many Microsoft Direct3D® samples in the DirectX SDK use file-based meshes. However, the ProgressiveMesh sample is a good example of the basic code necessary for loading a mesh. The D3DX mesh loading functionality collapses the frame hierarchy of an .x file into one mesh.

The primary reason to use progressive meshes is the call to **ID3DXPMesh::SetNumVertices** for the mesh.

For other samples, the basic D3DX mesh functionality is wrapped in a common class CD3DMesh. To keep the frame hierarchy, you can use the common class CD3DFile.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You

can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

RTPatch Sample

Description

The RTPatch sample shows how uses patches in Direct3D.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\RTPatch

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
UP ARROW	Increase number of patches per segment.
DOWN ARROW	Decrease number of patches per segment.
W	Toggles the wireframe.

Programming Notes

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

SkinnedMesh Sample

Description

The SkinnedMesh sample shows how to load and render a skinned mesh.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\SkinnedMesh

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

ShadowVolume Sample

Description

The ShadowVolume sample uses stencil buffers to implement real-time shadows. In the sample, a complex object is rendered and used as a shadow caster to cast real-time shadows on itself and on the terrain below.

Stencil buffers are a depth-buffer technique that can be updated as geometry is rendered, and used again as a mask for drawing more geometry. Common effects include mirrors, shadows (an advanced technique), dissolves, and so on.

Not all cards support all features for stencil-buffer techniques. Some hardware has no, or limited, stencil-buffer support. For more information on stencil buffers, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\StencilBuffer\ShadowVolume

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Creating real-time shadows is a fairly advanced technique. In each frame, or as the geometry or lights in the scene are moved, an object called a shadow volume is computed. A shadow volume is a 3-D object that is the silhouette of the shadowcasting object, as protruded away from the light source.

In this sample, the 3-D object that casts shadows is a bi-plane. The silhouette of the plane is computed in each frame. This technique uses an edge-detection algorithm in which silhouette edges are found. This can be done because the normals of adjacent polygons will have opposing normals with respect to the light vector. The resulting edge list (the silhouette) is protruded into a 3-D object away from the light source. This 3-D object is known as the shadow volume, as every point inside the volume is inside a shadow.

Next, the shadow volume is rendered into the stencil buffer twice. First, only forward-facing polygons are rendered, and the stencil-buffer values are incremented each time. Then the back-facing polygons of the shadow volume are drawn, decrementing values in the stencil buffer. Normally, all incremented and decremented values cancel each other out. However, because the scene was already rendered with normal geometry, in this case the plane and the terrain, some pixels fail the zbuffer test as the shadow volume is rendered. Any values left in the stencil buffer correspond to pixels that are in the shadow.

Finally, these remaining stencil-buffer contents are used as a mask, as a large all-encompassing black quad is alpha-blended into the scene. With the stencil buffer as a mask, only pixels in shadow are darkened

This sample uses common Microsoft DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

StencilDepth Sample

Description

The StencilDepth sample uses stencil buffers to display the depth complexity of a scene. The depth complexity of a scene is defined as the average number of times each pixel is rendered.

Stencil buffers are a depth-buffer technique that can be updated as geometry is rendered, and used again as a mask for drawing more geometry. Common effects include mirrors, shadows (an advanced technique), dissolves, and so on.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\StencilBuffer\StencilDepth

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Displaying depth complexity is a valuable tool to analyze the performance of a scene. Scenes with high amounts of overdraw can benefit from some scene optimization such as sorting the geometry in a front-to-back order.

Not all cards support all features for stencil-buffer techniques. Some hardware has no, or limited, stencil buffer-support. For more information on stencil buffers, refer to the Microsoft® DirectX® SDK documentation.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

StencilMirror Sample

Description

The StencilMirror sample uses stencil buffers to implement a mirror effect. In the sample, a watery terrain scene is rendered with the water reflecting a helicopter that flies above.

Stencil buffers are a depth-buffer technique that can be updated as geometry is rendered, and used again as a mask for drawing more geometry. Common effects include mirrors, shadows (an advanced technique), dissolves, and so on.

Not all cards support all features for stencil buffer-techniques. Some hardware has no, or limited, stencil buffer-support. For more information on stencil buffers, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\StencilBuffer\StencilMirror

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

In this sample, a stencil buffer is used to create the effect of a reflection coming off the water. The geometry of the water is rendered into the stencil buffer. Then the stencil buffer is used as a mask to render the scene again, this time with the geometry translated and rendered upside down, to appear as if it were reflected in the mirror.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

Text3D Sample

Description

The Text3D sample shows how to draw 2-D text in a 3-D scene. This is useful for displaying statistics or game menus, and so on.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\Text3D

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

This sample uses the common class, CD3DFont, to display 2-D text in a 3-D scene. The source code for the class is of most interest to this sample. The class uses GDI to load a font and output each letter to a bitmap. That bitmap, in turn, is used to create a texture.

When the CD3DFont class's **DrawText** function is called, a vertex buffer is filled with polygons that are textured using the font texture created as mentioned above. The polygons may be drawn as a 2-D overlay—useful, for example, for printing statistics—or truly integrated in the 3-D scene.

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

VertexBlend Sample

Description

The VertexBlend sample demonstrates a technique called vertex blending, also known as surface skinning. It displays a file-based object that is made to bend in various spots.

Vertex blending is used for creating effects such as smooth joints and bulging muscles in character animations.

Not all cards support all features for vertex blending. For more information on vertex blending, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\VertexBlend

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.
CTRL-V	Switches between a custom vertex shader and Direct3D vertex blending.

Programming Notes

Vertex blending requires each vertex to have an associated blend weight. Multiple world transforms are set up using **SetTransformState** and the blend weights determine how much contribution each world matrix has when positioning each vertex.

In this sample, a mesh is loaded using the common helper code. Note how a custom vertex and a custom FVF is declared and used to build the mesh; see the **SetFVF** call for the mesh object. Without using the mesh helper code, the technique is the same. Create a vertex buffer full of vertices that have a blend weight, and use the appropriate FVF.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

VertexShader Sample

Description

The VertexShader sample shows some effects that can be achieved using vertex shaders. Vertex shaders use a set of instructions, executed by the 3-D device on a per-vertex basis, that can affect the properties of the vertex—positions, normal, color, texture coordinates, and so on—in interesting ways.

Not all cards support all features for vertex shaders. For more information on vertex shaders, refer to the Microsoft® DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\VertexShader

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
W, S	Moves forward and back.
E, Q	Turns left and right.
A, Z	Rotates up and down.
ARROW KEYS	Slides left, right, up, and down.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Programming vertex shaders is not a trivial task. Refer to the Microsoft DirectX® SDK documentation for more information.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

VolumeTexture Sample

Description

The VolumeTexture sample illustrates how to use the new volume textures in Microsoft® Direct3D®. Normally, a texture is thought of as a 2-D image, which has a width and a height and whose texels are addressed with two coordinate, tu and tv. Volume textures are the 3-D counterparts, with a width, height, and depth, are addressed with three coordinates, tu, tv, and tw.

You can use volume textures for interesting effects such as patchy fog, explosions, and so on.

Not all cards support all features for volume textures. For more information on volume textures, refer to the Microsoft DirectX® SDK documentation.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\VolumeTexture

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Volume textures are no more difficult to use than 3-D textures. In this sample source code, note the vertex declaration, which has a third texture coordinate; texture creation, which also takes a depth dimension; and texture locking, again with the third dimension. The 3-D rasterizer interpolates texel values much as for 2-D textures.

This sample uses common DirectX code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

Water Sample

Description

The Water sample illustrates using D3DX techniques stored in shader files.

The sample shows a square pond inside a building, with rippling water effects including water caustics.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\Water

Executable: (SDK Root)\Samples\Multimedia\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented. You can use menu commands for the same controls.

Key	Action
ENTER	Starts and stops the scene.
SPACEBAR	Advances the scene by a small increment.
W, S	Moves forward and back.
E, Q	Turns left and right.
A, Z	Rotates up and down.
ARROW KEYS	Slides left, right, up, and down.
D	Starts a water drop.
PAGE DOWN	Displays next technique.
PAGE UP	Displays previous technique.
SHIFT+PAGE DOWN	Displays next technique with no validation.
SHIFT+PAGE UP	Displays previous technique with no validation.
F1	Shows Help or available commands.
F2	Prompts the user to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The D3DX shaders technique is the focus of this sample.

This sample uses common Microsoft® DirectX® code that consists programming elements such as helper functions. This code is shared with other samples in the DirectX SDK. You can find the common headers and source code in (SDK Root) \Samples\Multimedia\Common.

DirectX Graphics Visual Basic Samples

The following sample applications demonstrate the use and capabilities of the Microsoft® Direct3D® and Direct3DX application programming interfaces.

- AnimKeys Sample
- AutoParts Sample
- BarGraph Sample
- Billboard Sample
- Dolphin Sample
- Donuts Sample
- DXTex Tool
- PixelShader Sample
- PointSprites Sample
- ScatterGraph Sample
- SkinnedMesh Sample
- VertexBlend Sample
- VertexShader Sample

See Also

DirectX Graphics Visual Basic Tutorials

AnimKeys Sample

Description

The AnimKeys sample illustrates how to use the Microsoft® Direct3D® framework to load and play key-framed animation with an .x file. The animation information is contained in the .x file.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\AnimKeys

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented.

Key	Action
F2	Prompts you to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The last argument passed to **D3DUtil_LoadFromFile** is a CD3DAnimation class that is the parent to any animations that are in the .x file. Subsequently **Animation.SetTime** can be used to pose the model.

This sample uses common Microsoft DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

AutoParts Sample

Description

The AutoParts sample illustrates the use of picking against a 3-D database.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Autoparts

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

Choose **Click here** in the tree view on the left side to load the engine model. You can rotate the model by holding the mouse button and dragging. When you click on a part, its description appears in the lower-left corner.

The **Add and Remove from Invoice** button manages the invoice list, but the **Order** button has no function.

Programming Notes

Each object in the Engine model is named. These names are unique and can be used to cross-reference a database. In this example, a custom text database is used so as not to require installing MDAC. From that database, more detailed information, such as price and part number, is gathered on the part.

This sample uses common Microsoft® DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

BarGraph Sample

Description

The BarGraph sample describes how you can use Microsoft® Direct3D® for graphic visualization. It uses the **RenderToSurface** features of the D3DX framework to render text and bitmaps dynamically.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\AnimKeys

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

Right-click on the sample window to display a shortcut menu. From this menu you can load new data from a .csv (comma-delimited) file. You can export a .csv file from Microsoft Excel or any spreadsheet program.

The file must be formatted so that the first row and columns are headers that title the data. A header can contain the tag TEXTURE:*filename*.bmp to indicate that it contains a picture. The rest of the data must be numeric. See Bargraphdata.csv in (SDK ROOT)\Samples\Multimedia\VBSamples\Media for an example.

Holding the mouse button and dragging rotates the graph.

The following table lists the keys that are implemented.

Key	Action
RIGHT ARROW	Moves the camera right.
LEFT ARROW	Moves the camera left.
UP ARROW	Moves the camera up.
DOWN ARROW	Moves the camera down.
W	Moves the camera forward.
S	Moves the camera backward.

E	Rotates the camera right.
Q	Rotates the camera left.
A	Rotates the camera up.
Z	Rotates the camera down.

Programming Notes

This sample uses common Microsoft DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

Billboard Sample

Description

The Billboard sample illustrates the billboarding technique. Rather than rendering complex 3-D models, such as a high-polygon tree model, billboarding renders a 2-D image of the model and rotates it so that it always faces the eyepoint. This technique is commonly used to render trees, clouds, smoke, explosions, and so on. For more information, see Billboarding.

In the Billboard sample, a camera flies around a 3-D scene with a tree-covered hill. The trees look like 3-D objects, but they are actually 2-D billboarded images that are rotated toward the eyepoint. The hilly terrain and the skybox, a six-sided cube containing sky textures, are objects loaded from .x files. They are used for visual effect and are unrelated to the billboarding technique.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Billboard

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented.

Key	Action
F2	Prompts you to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The billboarding technique is the focus of this sample. The camera is moved in each frame, so that the viewpoint changes accordingly. As the viewpoint changes, a rotation matrix is generated to rotate the billboards about the y-axis so that they face the new viewpoint. The computation of the billboard matrix occurs in the **FrameMove** function. The trees are also sorted in that function, as required for proper alpha blending, because billboards typically have some transparent pixels. The trees are rendered from a vertex buffer in the **DrawTrees** function.

The billboards in this sample are constrained to rotate about the y-axis only. Otherwise the tree trunks would not appear to be fixed to the ground. In a 3-D flight simulation or space shooter, for effects like explosions, billboards are typically not constrained to one axis.

This sample uses common Microsoft DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

Dolphin Sample

Description

The Dolphin sample shows an underwater scene of a dolphin swimming, with caustic effects on the dolphin and sea floor. The dolphin is animated using a technique called tweening. The underwater effect uses fog, and the water caustics use an animated set of textures.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Dolphin

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented.

Key	Action
F2	Prompts you to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Several things are happening in this sample. First, the use of fog gives an underwater effect. The fog parameters are set in the **InitDeviceObjects** function.

The water caustics are achieved by animating a set of 32 textures (Caust00.tga through Caust31.tga). The caustics can be blended into the scene using multitexturing or multipass blending techniques. Because the bottom of the dolphin should not have caustic effects, a separate pass is done in which ambient light is removed and the dolphin is lit from above, and then the diffuse color is blended with the caustic texture.

Using tweening, the dolphin model's vertices are linearly blended from multiple sets of vertices. The source models for these other sets of vertices are loaded from Dolphin_group.x, which consists of the dolphin model in three positions. For each frame, a destination mesh is generated by blending some combination of the positions and normals from these meshes together.

For more information on tweening, see Vertex Tweening.

This sample uses common Microsoft® DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

Donuts Sample

Description

The Donuts sample illustrates how to use Microsoft® Direct3D® to create a two-dimensional sprite engine.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Donuts

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented.

Key	Action
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The background is rendered with two TLVertex (screen space) triangles. The sprites are defined as an array of user-defined type that is a container for sprite properties such as position, direction, speed, and size. Each sprite is rendered as two TLVertex triangles in each frame.

This sample uses common Microsoft DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

DXTex Tool

Description

The DXTex tool enables you to create texture maps that use the new DXTn compression formats. Creating a DXTn-compressed texture is not difficult. However, Microsoft® Direct3D® can do the conversion for you when using the **Direct3DTexture8** object. Advanced developers can write tools that meet their specific needs, but the DXTex tool provides useful basic functionality.

Functionality

- Opens .bmp and DirectDrawSurface (DDS) files. For more information, see DDS File Format below.
- Opens .bmp files as alpha channel, either explicitly or implicitly, using a <*filename*>_a.bmp naming convention.
- Saves textures in DDS format.
- Supports conversion to all five DXTn compression formats.
- Supports generation of mipmaps, using a box filter.
- Supports visualization of alpha channel as a grayscale image or through a user-selectable background color.
- Supports easy visual comparison of image quality between formats.

Path

Source: (SDK Root)\Samples\Multimedia\Direct3D\DXTex

Executable: (SDK Root)\Bin\DXUtils

User Interface

DxTex uses a fairly traditional user interface (UI) in that each texture map is a document, and several documents can be open at a time. However, each document can hold the texture in either one or two formats at once while the document is open

in DxTex. For example, you can import a .bmp file, which automatically creates a 32-bit ARGB texture. You can then convert the texture to DXT1 format. The document now has the image open in both formats, and you can switch between the formats by clicking the window or choosing **Original View** or **New View** on the **View** menu. This makes it easy for you to observe any artifacts introduced by image compression, and to try several compression formats without progressive degradation of the image. For example, if this technique is not used and you convert an image from ARGB to DXT1, all but one bit of alpha is lost.

If you then decide to convert to DXT2, there is still only two alpha levels. Using the DxTex system, the second conversion is done from the original ARGB format, and the DXT2 image contains all 16 levels of alpha supported by DXT2. When the image is saved, the original format is discarded, and only the new format is stored.

Keep in mind the following when using the DxTex interface.

- If no format conversion has been requested since the document was opened, the one format is the original format—that is, the format in which the image will be written when the document is saved.
- If format conversion has been requested since the document was opened, the converted format is the new format—that is, the format in which the image will be written when the document is saved.
- If a second format conversion is requested, the new format from the first conversion is replaced with the second format. The original format is unchanged.
- Generating mipmaps applies to both formats simultaneously.
- If the original format has multiple mipmaps, and conversion to a new format is requested, the new format automatically has multiple mipmaps as well.

Performance

DxTex uses the Microsoft Direct3D Reference Rasterizer to draw the textures, regardless of what 3-D hardware is available. So with larger textures (greater than 256x256 pixels), the program may be somewhat slow, depending on your CPU speed.

DSS File Format

The DxTex native file format is called the DDS file format because it encapsulates the information in a DirectDrawSurface. It has the following format.

```
Magic As Long '0x20534444, or "DDS"
ddsd As DDSURFACEDESC2    ' Provides information about the surface
                          ' format.
Data1 As Any              ' First element of an array of surface data
                          ' for the main surface.
[Data2 As Any...]         ' First element of an array of surface data
                          ' for attached surfaces follows.
```

This format is easy to read and write to, and it supports features such as alpha and multiple mip levels, as well as DXTn compression. The Compress sample application in the DirectX SDK demonstrates how to properly import a DDS file.

It is not necessary to use the DDS format in order to use DXTn-compressed textures, or vice-versa, but the two work well together.

Mipmaps

Mipmapping is a technique that improves image quality and reduces texture memory bandwidth by providing prefiltered versions of the texture image at multiple resolutions.

To generate mipmaps in DxTex, the width and height of the source image must both be powers of 2. To do this, click **Generate Mip Maps** on the **Format** menu. Filtering is done through a simple box filter. That is, the four nearest pixels are averaged to produce each destination pixel.

Alpha

Many texture formats include an alpha channel, which provides opacity information at each pixel. DxTex fully supports alpha in textures. When importing a .bmp file, if there are two files of the same size, and one of the file names ends in "_a" (for example, Sample.bmp and Sample_a.bmp), the second file will be loaded as an alpha channel. The blue channel of the second .bmp is stored in the alpha channel. Once a document is open, you can explicitly load a .bmp file as the alpha channel by, on the **File** menu, clicking **Open As Alpha Channel**.

To view the alpha channel directly without the RGB channels, click **Alpha Channel Only** on the **View** menu. The alpha channel appears as a grayscale image. If no alpha channel has been loaded, every pixel has a full alpha channel and the image appears solid white when viewing "Alpha Channel Only". To turn off alpha channel viewing, click the **Alpha Channel Only** command a second time.

In the usual view, the effect of the alpha channel is visible because the window has a solid background color that shows through the texture where the alpha channel is less than 100 percent. You can change this background color by clicking the **View** menu, and then clicking **Change Background Color**. This choice does not affect the texture itself or the data that is written when the file is saved.

The DXT2 and DXT4 formats use premultiplied alpha. This means that the red, green, and blue values stored in the surface are already multiplied by the corresponding alpha value. Direct3D cannot copy from a surface that contains premultiplied alpha to one that contains non-premultiplied alpha. Therefore, some DxTex operations—such as Open as Alpha Channel, conversion to DXT3, and conversion to DXT5—are not possible on DXT2 and DXT4 formats. Supporting textures using these formats is difficult on Direct3D devices that do not support DXTn textures. This is because Direct3D cannot copy them to a traditional ARGB surface either, unless that ARGB surface uses premultiplied alpha as well, which is

rare. For this reason, you might find it easier to use DXT3 rather than DXT2, and DXT5 rather than DXT4 when possible.

Command Line Options

You can use command-line options to pass input files, an output file name, and processing options to DxTex. If an output file name is specified, the program exits automatically after writing the output file, and no user interface is presented.

```
dxtex [infilename] [-a alphaname] [-m] [DXT1|DXT2|DXT3|DXT4|DXT5] [outfilename]

infilename:          Name of the file to load. This can be a .bmp
                     or DDS file.

-a alphaname:        The next parameter is the name of a .bmp
                     file to load as the alpha channel. If no
                     alpha filename is specified, DxTex still
                     looks for a file named Infilename_a.bmp and,
                     if it exists, use that file as the alpha
                     channel.

-m:                  Mipmaps are generated.

DXT1|DXT2|DXT3|DXT4|DXT5: Compression format. If no format is
                     specified, the image is be in ARGB-8888.

outfilename:         Name of the destination file. If this option
                     is not specified, the user interface shows
                     the current file and all requested
                     operations. If an outfilename is specified,
                     the application exits after saving the
                     processed file without presenting a user
                     interface.
```

PixelShader Sample

Description

This sample shows some effects that can be achieved using pixel shaders. Each of the eight thumbnails is the result of using a different pixel shader to render a rectangle. Pixel shaders use a set of instructions, executed by the 3-D device on a per-pixel basis, that can affect the color of the pixel based on a variant of inputs. Pixel shaders can be used in place of the texture stage pipeline.

Not all cards support all features for pixel shaders. For more information on pixel shaders, see Pixel Shaders.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\PixelShader

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

Programming Notes

This sample uses common Microsoft® DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

PointSprites Sample

Description

The PointSprites sample shows how to use the new Microsoft® Direct3D® point sprites feature. A point sprite is a forward-facing, textured quad that is referenced only by (x,y,z) position coordinates. Point sprites are most often used for particle systems and related effects.

Not all cards support all features for point sprites. For more information on point sprites, see Point Sprites.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\PointSprites

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented.

Key	Action
F2	Prompts you to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Without the Direct3D support, point sprites can be implemented with four vertices that are oriented each frame towards the eyepoint, much like a billboard. With Direct3D, however, you can refer to each point sprite just by its center position and a radius. This saves on processor computation time and on bandwidth when uploading vertex information to the graphics card.

In this sample, a particle system is implemented using point sprites in which each particle is implemented using multiple alpha-blended point sprites, giving the particle a motion-blur effect.

This sample uses common Microsoft DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

ScatterGraph Sample

Description

The ScatterGraph sample describes how you can use Microsoft® Direct3D® for graphic visualization. It uses the **RenderToSurface** features of the D3DX framework to render text and bitmaps dynamically.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\AnimKeys

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

Right-click the sample's interface to display a shortcut menu. The following table list the available menu commands.

Command	Action
Load Data From File	Loads new data from a .csv (comma-delimited) file. A .csv file can be exported from Microsoft Excel or any spreadsheet program.
Reset Orientation	Resets the viewpoint to a known state.
Show Connecting Lines	Connects the data points if the order of the data is important.
Show Height Lines	Makes it easier to see the y-value in comparison to the other values.
Show Foot Lines	Makes it easier to see the x-z relationship.
Show Base Plane	Shows the plane where y=0.
Auto Rotate	Turns rotation on and off.

The file must be formatted so that the first row and columns are headers that title the data. A header can contain the tag TEXTURE:*filename*.bmp to indicate that it contains a picture. The rest of the data must be numeric. See Bargraphdata.csv in (SDK ROOT)\Samples\Multimedia\VBSamples\Media for an example.

Holding the mouse button and dragging rotates the graph.

The following table lists the keys that are implemented.

Key	Action
RIGHT ARROW	Moves the camera right.
LEFT ARROW	Moves the camera left.
UP ARROW	Moves the camera up.
DOWN ARROW	Moves the camera down.
W	Moves the camera forward.
S	Moves the camera backward.
E	Rotates the camera right.
Q	Rotates the camera left.
A	Rotates the camera up.
Z	Rotates the camera down.

Programming Notes

This sample uses common Microsoft DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

SkinnedMesh Sample

Description

The SkinnedMesh sample illustrates how to use the Microsoft® Direct3D® framework to load an .x file with skinning and animation information in it.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\SkinnedMesh

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented.

Key	Action
F2	Prompts you to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

The last argument passed to **D3DUtil_LoadFromFileAsSkin** is a CD3DAnimation class that is the parent to any animations that are found in the .x file.

Animation.SetTime must be called but does not pose the model.

Character.UpdateFrames computes the matrices for all joints on the character.

Character.RenderSkin renders the character using the loaded skin.

This sample uses common Microsoft DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

The modeling exporters in the Extras directory of the DirectX SDK can export to .x files with skinning information.

VertexBlend Sample

Description

The VertexBlend sample demonstrates a technique called vertex blending, also known as surface skinning. It displays a file-based object that is made to bend at various points. Surface skinning is an impressive technique used for effects such as smooth joints and bulging muscles in character animations.

Not all display cards support all features for vertex blending. For more information on vertex blending, see Indexed Vertex Blending.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\VertexBlend

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented.

Key	Action
F2	Prompts you to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

Vertex blending requires each vertex to have an associated blend weight. Multiple world transforms are set up using **SetTransformState**, and the blend weights determine how much contribution each world matrix has when positioning each vertex.

In this sample, a mesh is loaded using the common helper code. Note how a custom vertex and a custom FVF is declared and used to build the mesh; see the **SetFVF** call for the mesh object. Without using the mesh helper code, the technique is the same. Create a vertex buffer full of vertices that have a blend weight, and use the appropriate FVF.

This sample uses common Microsoft® DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

VertexShader Sample

Description

The VertexShader sample shows some effects that you can achieve using vertex shaders. Vertex shaders use a set of instructions, executed by the 3-D device on a per-vertex basis, that can affect the properties of the vertex—positions, normal, color, texture coordinates, and so on—in interesting ways.

Not all cards support all features for vertex shaders. For more information on vertex shaders, see Vertex Shaders.

Path

Source: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\VertexShader

Executable: (SDK Root)\Samples\Multimedia\VBSamples\Direct3D\Bin

User's Guide

The following table lists the keys that are implemented.

Key	Action
F2	Prompts you to select a new rendering device or display mode.
ALT+ENTER	Toggles between full-screen and windowed modes.
ESC	Exits the application.

Programming Notes

This sample uses common Microsoft® DirectX® code that consists of programming elements such as helper functions. This code is shared with other samples on the DirectX SDK. You can find the common source code in (SDK Root) \Samples\Multimedia\VBSamples.

Direct3D C/C++ Reference

This section contains reference information for the API elements provided by Microsoft® Direct3D®. Reference material is divided into the following categories.

- Interfaces
- Functions
- Macros
- Vertex Shader Declarator Macros
- Structures
- Enumerated Types
- Other Types
- Texture Argument Flags
- Flexible Vertex Format Flags
- Four-Character Codes (FOURCC)
- Return Values

Interfaces

This section contains reference information for the COM interfaces provided by Microsoft® Direct3D®. The following interfaces are used with Direct3D.

- **IDirect3D8**
- **IDirect3DBaseTexture8**
- **IDirect3DCubeTexture8**
- **IDirect3DDevice8**
- **IDirect3DIndexBuffer8**
- **IDirect3DResource8**
- **IDirect3DSurface8**
- **IDirect3DSwapChain8**
- **IDirect3DTexture8**
- **IDirect3DVertexBuffer8**

- **IDirect3DVolume8**
- **IDirect3DVolumeTexture8**

IDirect3D8

Applications use the methods of the **IDirect3D8** interface to create Microsoft® Direct3D® objects and set up the environment. This interface includes methods for enumerating and retrieving capabilities of the device.

The **IDirect3D8** interface is obtained by calling the **Direct3DCreate8** function.

The methods of the **IDirect3D8** interface can be organized into the following groups.

Creation	**CreateDevice**
Enumeration	**EnumAdapterModes**
Information	**GetAdapterCount**
	GetAdapterDisplayMode
	GetAdapterIdentifier
	GetAdapterModeCount
	GetAdapterMonitor
	GetDeviceCaps
Registration	**RegisterSoftwareDevice**
Verification	**CheckDepthStencilMatch**
	CheckDeviceFormat
	CheckDeviceMultiSampleType
	CheckDeviceType

The **IDirect3D8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3D8** and **PDIRECT3D8** types are defined as pointers to the **IDirect3D8** interface.

```
typedef struct IDirect3D8 *LPDIRECT3D8, *PDIRECT3D8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

Direct3DCreate8

IDirect3D8::CheckDepthStencilMatch

Determines whether or not a depth-stencil format is compatible with a render target format in a particular display mode.

```
HRESULT CheckDepthStencilMatch(
  UINT Adapter,
  D3DDEVTYPE DeviceType,
  D3DFORMAT AdapterFormat,
  D3DFORMAT RenderTargetFormat,
  D3DFORMAT DepthStencilFormat
);
```

Parameters

Adapter
[in] Ordinal number denoting the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter.

DeviceType
[in] Member of the **D3DDEVTYPE** enumerated type, identifying the device type.

AdapterFormat
[in] Member of the **D3DFORMAT** enumerated type, identifying the format of the display mode into which the adapter will be placed.

RenderTargetFormat
[in] Member of the **D3DFORMAT** enumerated type, identifying the format of the render target surface to be tested.

DepthStencilFormat
[in] Member of the **D3DFORMAT** enumerated type, identifying the format of the depth-stencil surface to be tested.

Return Values

If the depth-stencil format is compatible with the render target format in the display mode, this method returns D3D_OK.

D3DERR_INVALIDCALL can be returned if one or more of the parameters is invalid. If a depth-stencil format is not compatible with the render target in the display mode, then this method returns D3DERR_NOTAVAILABLE.

Remarks

This method is provided to enable applications to work with hardware requiring that certain depth formats can only work with certain render target formats.

The following code fragment shows how you could use **CheckDeviceFormat** to validate a depth stencil format.

```
BOOL IsDepthFormatOk( D3DFORMAT DepthFormat, D3DFORMAT AdapterFormat,
                      D3DFORMAT BackBufferFormat ) {
    // Verify that the depth format exists.
    HRESULT hr = pD3D->CheckDeviceFormat( D3DADAPTER_DEFAULT,
                                          D3DDEVTYPE_HAL,
                                          AdapterFormat,
                                          D3DUSAGE_DEPTHSTENCIL,
                                          D3DRTYPE_SURFACE,
                                          DepthFormat);

    if( FAILED( hr ) ) return FALSE;

    // Verify that the depth format is compatible.
    hr = pD3D->CheckDepthStencilMatch( D3DADAPTER_DEFAULT,
                                       D3DDEVTYPE_HAL,
                                       AdapterFormat,
                                       BackBufferFormat,
                                       DepthFormat);
    return SUCCEEDED( hr );
}
```

The preceding call will return FALSE if *DepthFormat* cannot be used in conjunction with *AdapterFormat* and *BackBufferFormat*.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3D8::CheckDeviceFormat

Determines whether a surface format is available as a specified resource type and can be used as a texture, depth-stencil buffer, or render target, or any combination of the three, on a device representing this adapter.

HRESULT CheckDeviceFormat(

```
  UINT Adapter,
  D3DDEVTYPE DeviceType,
  D3DFORMAT AdapterFormat,
  DWORD Usage,
  D3DRESOURCETYPE RType,
  D3DFORMAT CheckFormat
);
```

Parameters

Adapter

[in] Ordinal number denoting the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter. This method returns D3DERR_INVALIDCALL when this value equals or exceeds the number of display adapters in the system.

DeviceType

[in] Member of the **D3DDEVTYPE** enumerated type, identifying the device type.

AdapterFormat

[in] Member of the **D3DFORMAT** enumerated type, identifying the format of the display mode into which the adapter will be placed.

Usage

[in] Requested usage for a surface of the *CheckFormat*. One or more of the following flags can be specified.

D3DUSAGE_DEPTHSTENCIL
: Set to indicate that the surface can be used as a depth-stencil surface.

D3DUSAGE_RENDERTARGET
: Set to indicate that the surface can be used as a render target. In cases where the *RType* parameter value unambiguously implies a specific usage, the application can leave the *Usage* parameter as 0.

RType

[in] A member of the **D3DRESOURCETYPE** enumerated type, specifying the resource type requested for use with the queried format.

CheckFormat

[in] Member of the **D3DFORMAT** enumerated type. Indicates the format of the surfaces which may be used, as defined by *Usage*.

Return Values

If the format is compatible with the specified device for the requested usage, this method returns D3D_OK.

D3DERR_INVALIDCALL is returned if *Adapter* equals or exceeds the number of display adapters in the system, or if *DeviceType* is unsupported. This method returns

D3DERR_NOTAVAILABLE if the format is not acceptable to the device for this usage.

Remarks

A typical use of **CheckDeviceFormat** is to verify the existence of a particular depth-stencil surface format. See Selecting a Device for more detail on the enumeration process. The following code fragment shows how you could use **CheckDeviceFormat** to verify the existence of a depth-stencil format.

```
BOOL IsDepthFormatExisting( D3DFORMAT DepthFormat, D3DFORMAT AdapterFormat ) {
    HRESULT hr = pD3D->CheckDeviceFormat( D3DADAPTER_DEFAULT,
                                          D3DDEVTYPE_HAL,
                                          AdapterFormat,
                                          D3DUSAGE_DEPTHSTENCIL,
                                          D3DRTYPE_SURFACE,
                                          DepthFormat);

    return SUCCEEDED( hr );
}
```

The preceding call will return FALSE if *DepthFormat* does not exist on the system.

Another typical use of **CheckDeviceFormat** would be to verify if textures existing in particular surface formats can be rendered, given the current display mode. The following code fragment shows how you could use **CheckDeviceFormat** to verify that texture formats are compatible with specific back buffer formats.

```
BOOL IsTextureFormatOk( D3DFORMAT TextureFormat, D3DFORMAT AdapterFormat ) {
    HRESULT hr = pD3D->CheckDeviceFormat( D3DADAPTER_DEFAULT,
                                          D3DDEVTYPE_HAL,
                                          AdapterFormat,
                                          0,
                                          D3DRTYPE_TEXTURE,
                                          TextureFormat);

    return SUCCEEDED( hr );
}
```

The preceding call will return FALSE if *TextureFormat* cannot be used to render textures while the adapter surface format is *AdapterFormat*.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3D8::CheckDeviceMultiSampleType

Determines if a multisampling technique is available on this device.

```
HRESULT CheckDeviceMultiSampleType(
  UINT Adapter,
  D3DDEVTYPE DeviceType,
  D3DFORMAT SurfaceFormat,
  BOOL Windowed,
  D3DMULTISAMPLE_TYPE MultiSampleType
);
```

Parameters

Adapter
[in] Ordinal number denoting the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter. This method returns FALSE when this value equals or exceeds the number of display adapters in the system. See Remarks.

DeviceType
[in] Member of the **D3DDEVTYPE** enumerated type, identifying the device type.

SurfaceFormat
[in] Member of the **D3DFORMAT** enumerated type that specifies the set of multisampling types requested for the surface. See Remarks.

Windowed
[in] Boolean value. Specify TRUE to inquire about windowed multisampling, and specify FALSE to inquire about full-screen multisampling.

MultiSampleType
[in] Member of the **D3DMULTISAMPLE_TYPE** enumerated type, identifying the multisampling technique to test.

Return Values

If the device can perform the specified multisampling method, this method returns D3D_OK.

D3DERR_INVALIDCALL is returned if the *Adapter* or *MultiSampleType* parameters are invalid. This method returns D3DERR_NOTAVAILABLE if the queried multisampling technique is not supported by this device. D3DERR_INVALIDDEVICE is returned if *DeviceType* does not apply to this adapter.

Remarks

This method is intended for use with both render-target and depth-stencil surfaces because you must create both surfaces multisampled if you want to use them together.

The following code fragment shows how you could use **CheckDeviceMultiSampleType** to test for devices that support a specific multisampling method.

```
if( SUCCEEDED(pDevice->CheckDeviceMultiSampleType( pCaps->AdapterOrdinal,
                        pCaps->DeviceType, pMode->Format,
                        FALSE, D3DMULTISAMPLE_3_SAMPLES ) ) )
    return S_OK;
```

The preceding code will return S_OK if the device supports the full-screen D3DMULTISAMPLE_3_SAMPLES multisampling method with the surface format.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3D8::CheckDeviceType

Verifies whether or not a certain device type can be used on this adapter and expect hardware acceleration using the given formats.

```
HRESULT CheckDeviceType(
  UINT Adapter,
  D3DDEVTYPE CheckType,
  D3DFORMAT DisplayFormat,
  D3DFORMAT BackBufferFormat,
  BOOL Windowed
);
```

Parameters

Adapter
[in] Ordinal number denoting the display adapter to enumerate. D3DADAPTER_DEFAULT is always the primary display adapter. This method returns D3DERR_INVALIDCALL when this value equals or exceeds the number of display adapters in the system.

CheckType
[in] Member of the **D3DDEVTYPE** enumerated type, indicating the device type to check.

DisplayFormat

[in] Member of the **D3DFORMAT** enumerated type, indicating the format of the adapter display mode for which the device type is to be checked. For example, some devices will operate only in 16-bits-per-pixel modes.

BackBufferFormat

[in] Member of the **D3DFORMAT** enumerated type, indicating the format of the back buffer to be tested

Windowed

[in] Value indicating whether the device type will be used in full-screen or windowed mode. If set to TRUE, the query is performed for windowed applications; otherwise, this value should be set FALSE.

Return Values

If the device can be used on this adapter, D3D_OK is returned.

D3DERR_INVALIDCALL is returned if *Adapter* equals or exceeds the number of display adapters in the system. This method returns D3DERR_INVALIDDEVICE if *CheckType* specified a device that does not exist. D3DERR_NOTAVAILABLE is returned if either surface format is not supported, or if hardware acceleration is not available for the specified formats.

Remarks

The most important device type that may not be present is D3DDEVTYPE_HAL. D3DDEVTYPE_HAL requires hardware acceleration. Applications should use **CheckDeviceType** to determine if the needed hardware and drivers are present on the system. The other device that may not be present is D3DDEVTYPE_SW. This device type represents a pluggable software device that was registered using **IDirect3D8::RegisterSoftwareDevice**.

Applications should not specify a *DisplayFormat* that contains an alpha channel. This will result in a failed call. Note that an alpha channel may be present in the back buffer, but the two display formats must be identical in all other respects. For example, if *DisplayFormat* is D3DFMT_X1R5G5B5, valid values for *BackBufferFormat* include D3DFMT_X1R5G5B5 and D3DFMT_A1R5G5B5, but exclude D3DFMT_R5G6B5.

The following code fragment shows how you could use **CheckDeviceType** to test whether or not a certain device type can be used on this adapter.

```
if(SUCCEEDED(pD3Device->CheckDeviceType(D3DADAPTER_DEFAULT,
                        D3DDEVTYPE_HAL,
                        DisplayFormat,
                        BackBufferFormat,
                        bIsWindowed)))
    return S_OK;

// There is no HAL on this adapter using this render target format.
// Try again, using another format.
```

The preceding code will return S_OK if the device can be used on the default adapter, with the specified surface format.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3D8::CreateDevice

Creates a device to represent the display adapter.

```
HRESULT CreateDevice(
  UINT Adapter,
  D3DDEVTYPE DeviceType,
  HWND  hFocusWindow,
  DWORD BehaviorFlags,
  D3DPRESENT_PARAMETERS* pPresentationParameters,
  IDirect3DDevice8** ppReturnedDeviceInterface
);
```

Parameters

Adapter
[in] Ordinal number that denotes the display adapter. D3DADAPTER_DEFAULT is always the primary display adapter.

DeviceType
[in] Member of the **D3DDEVTYPE** enumerated type. Denotes the desired device type. If the desired device type is not available, the method will fail.

hFocusWindow
[in] Window handle to which focus belongs for this Microsoft® Direct3D® device. The window specified must be a top-level window for full-screen.

BehaviorFlags
[in] A combination of one or more flags that control global behaviors of the Direct3D device.

D3DCREATE_FPU_PRESERVE
Indicates that the application needs either double precision FPU or FPU exceptions enabled. Direct3D sets the FPU state each time it is called. Setting the flag will reduce Direct3D performance.

D3DCREATE_HARDWARE_VERTEXPROCESSING
Specifies hardware vertex processing. See Remarks.

D3DCREATE_MIXED_VERTEXPROCESSING
Specifies mixed (both software and hardware) vertex processing. See Remarks.

D3DCREATE_MULTITHREADED
Indicates that the application requests Direct3D to be multithread safe. This makes Direct3D take its global critical section more frequently, which can degrade performance.

D3DCREATE_PUREDEVICE
Specifies that Direct3D does not support Get* calls for anything that can be stored in state blocks. It also tells Direct3D not to provide any emulation services for vertex processing. This means that if the device does not support vertex processing, then the application can use only post-transformed vertices.

D3DCREATE_SOFTWARE_VERTEXPROCESSING
Specifies software vertex processing. See Remarks.

pPresentationParameters
[in, out] Pointer to a **D3DPRESENT_PARAMETERS** structure, describing the presentation parameters for the device to be created. Calling **CreateDevice** can change the value of the **BackBufferCount** member of **D3DPRESENT_PARAMETERS**; the back buffer count is changed to reflect a corrected number of back buffers.

ppReturnedDeviceInterface
[out, retval] Address of a pointer to the returned **IDirect3DDevice8** interface, representing the created device.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

Remarks

This method returns a fully working device interface, set to the required display mode (or windowed), and allocated with the appropriate back buffers. The application only needs to create and set a depth buffer, if desired, to begin rendering.

This method should not be executed during the handling of WM_CREATE. Therefore, the application should never pass a window handle to Direct3D while handling WM_CREATE. DirectX 8.0 applications can expect messages to be sent to them during this call—that is, before this call is returned. Applications should take precautions not to call into Direct3D at this time. In addition, note that a call to create, release, or reset the device can only happen on the same thread as the window procedure of the focus window.

Note that D3DCREATE_HARDWARE_VERTEXPROCESSING, D3DCREATE_MIXED_VERTEXPROCESSING, and D3DCREATE_SOFTWARE_VERTEXPROCESSING are mutually exclusive flags, and that at least one of these vertex processing flags must be specified when calling **CreateDevice**.

Back buffers created as part of the device are only lockable if D3DPRESENTFLAG_LOCKABLE_BACKBUFFER is specified in the presentation parameters. (Multisampled back buffers and depth-surfaces are never lockable.)

The methods **IDirect3DDevice8::Reset**, **Release**, and **IDirect3DDevice8::TestCooperativeLevel** must be called from the same thread that used **CreateDevice** to create a device.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

Direct3DCreate8, **IDirect3DDevice8::Reset**, **D3DDEVICE_CREATION_PARAMETERS**

IDirect3D8::EnumAdapterModes

Enumerates the display modes of an adapter.

```
HRESULT EnumAdapterModes(
  UINT Adapter,
  UINT Mode,
  D3DDISPLAYMODE* pMode
);
```

Parameters

Adapter
[in] Ordinal number that denotes the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter.

Mode
[in] Ordinal number that denotes the mode to enumerate. D3DCURRENT_DISPLAY_MODE is a special value denoting the current desktop display mode on this adapter and cannot be specified for this method. See Remarks.

pMode
[in, out] Pointer to a **D3DDISPLAYMODE** structure, to be filled with information describing this mode. On error conditions, this value is zeroed.

Return Values

If the method succeeds, the return value is D3D_OK.

If *Adapter* or *Mode* is out of range, or *pMode* is invalid, this method returns D3DERR_INVALIDCALL.

Remarks

Some display drivers include refresh rate information that will greatly increase the number of modes. On older drivers, refresh rates of D3DADAPTER_DEFAULT are the only ones enumerated. Direct3D attempts to limit display modes and refresh rates based on the monitor capabilities, however there is still some risk that an enumerated mode will not be supported by the system's monitor. You may want to implement a display mode testing facility similar to that used by the Microsoft® Windows® Display Control Panel utility.

Use **IDirect3D8::GetAdapterModeCount** to determine the total number of modes supported by the system. Valid values for the *Mode* parameter range from 0 to the total number of modes minus 1.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3D8::GetAdapterModeCount

IDirect3D8::GetAdapterCount

Returns the number of adapters on the system.

```
UINT GetAdapterCount();
```

Parameters

None.

Return Values

A **UINT** value that denotes the number of adapters on the system at the time this **IDirect3D8** interface was instantiated.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3D8::GetAdapterIdentifier

IDirect3D8::GetAdapterDisplayMode

Retrieves the current display mode of the adapter.

```
HRESULT GetAdapterDisplayMode(
  UINT Adapter,
  D3DDISPLAYMODE* pMode
);
```

Parameters

Adapter
[in] Ordinal number that denotes the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter.

pMode
[in, out] Pointer to a **D3DDISPLAYMODE** structure, to be filled with information describing the current adapter's mode.

Return Values

If the method succeeds, the return value is D3D_OK.

If *Adapter* is out of range or *pMode* is invalid, this method returns D3DERR_INVALIDCALL.

Remarks

Note that a device that represents an adapter may have been reset into a particular mode, but this may not be the actual mode of the adapter. A device reset to a particular mode may be lost, and therefore it can not determine the current mode of the adapter. This function returns the adapter mode, not any particular device's desired mode.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3D8::GetAdapterIdentifier

Describes the physical display adapters present in the system when the **IDirect3D8** interface was instantiated.

```
HRESULT GetAdapterIdentifier(
  UINT Adapter,
  DWORD Flags,
  D3DADAPTER_IDENTIFIER8* pIdentifier
);
```

Parameters

Adapter
[in] Ordinal number that denotes the display adapter. D3DADAPTER_DEFAULT is always the primary display adapter. The minimum value for this parameter is 0, and the maximum value for this parameter is one less than the value returned by **IDirect3D8::GetAdapterCount**.

Flags
[in] Parameter that is typically set to zero. However, you can specify the following value.

D3DENUM_NO_WHQL_LEVEL
Forces the **WHQLLevel** member of the **D3DADAPTER_IDENTIFIER8** structure to be zero, meaning that no information is returned for the WHQL certification date. Setting this flag will avoid the one- or two-second time penalty incurred to determine the WHQL certification date.

pIdentifier
[in, out] Pointer to a **D3DADAPTER_IDENTIFIER8** structure to be filled with information describing this adapter. If *Adapter* is greater than or equal to the number of adapters in the system, this structure will be zeroed.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if *Adapter* is out of range, if *Flags* contains unrecognized parameters, or if *pIdentifier* is NULL or points to unwriteable memory.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3D8::GetAdapterCount

IDirect3D8::GetAdapterModeCount

Returns the number of display modes available on this adapter.

```
UINT GetAdapterModeCount(
  UINT Adapter
);
```

Parameters

Adapter
[in] Ordinal number that denotes the display adapter.
D3DADAPTER_DEFAULT is always the primary display adapter.

Return Values

This method returns the number of display modes on this adapter, or zero if *Adapter* is greater than or equal to the number of adapters on the system.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3D8::EnumAdapterModes

IDirect3D8::GetAdapterMonitor

Returns the handle of the monitor associated with the Microsoft® Direct3D® object.

```
HMONITOR GetAdapterMonitor(
  UINT Adapter
);
```

Parameters

Adapter
[in] Ordinal number that denotes the display adapter.
D3DADAPTER_DEFAULT is always the primary display adapter.

Return Values

Handle of the monitor associated with the Direct3D object.

Remarks

As shown in the following code fragment, which illustrates how to obtain a handle to the monitor associated with a given device, use **IDirect3DDevice8::GetDirect3D** to return the Direct3D enumerator from the device and use **IDirect3DDevice8::GetCreationParameters** to retrieve the value for *Adapter*.

```
if( FAILED( pDevice->GetCreationParameters(  &Parameters ) ) )
   return E_INVALIDCALL;

if( FAILED( pDevice->GetDirect3D(&pD3D) ) )
   return E_INVALIDCALL;

hMonitor = pD3D->GetAdapterMonitor(Parameters.AdapterOrdinal);

pD3D->Release();
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetCreationParameters, **IDirect3DDevice8::GetDirect3D**

IDirect3D8::GetDeviceCaps

Retrieves device-specific information about a device.

```
HRESULT GetDeviceCaps(
  UINT Adapter,
  D3DDEVTYPE DeviceType,
  D3DCAPS8* pCaps
);
```

Parameters

Adapter
[in] Ordinal number that denotes the display adapter.
D3DADAPTER_DEFAULT is always the primary display adapter.

DeviceType
[in] Member of the **D3DDEVTYPE** enumerated type. Denotes the device type.

pCaps
[out] Pointer to a **D3DCAPS8** structure to be filled with information describing the capabilities of the device.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_INVALIDDEVICE

D3DERR_OUTOFVIDEOMEMORY

Remarks

The application should not assume the persistence of vertex processing capabilities across Microsoft® Direct3D® device objects. The particular capabilities that a physical device exposes may depend on parameters supplied to **IDirect3D8::CreateDevice**. For example, the capabilities may yield different vertex processing capabilities before and after creating a Direct3D Device Object with hardware vertex processing enabled. For more information see the description of **D3DCAPS8**.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3D8::RegisterSoftwareDevice

Registers a pluggable software device with Microsoft® Direct3D®.

```
HRESULT RegisterSoftwareDevice(
  void* pInitializeFunction
);
```

Parameters

pInitializeFunction
[in] Pointer to the initialization function for the software device to be registered.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

Remarks

Software rasterization for Direct3D is provided by pluggable software devices, enabling applications to access a variety of software rasterizers through the Direct3D interfaces. Software devices are loaded by the application and registered with the Direct3D object, at which point a Direct3DDevice object can be created that will perform rendering with the software device.

Direct3D software devices communicate with Direct3D through an interface similar to the hardware device driver interface (DDI).

The Direct3D DDK provides the documentation and headers for developing pluggable software devices.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DBaseTexture8

Applications use the methods of the **IDirect3DBaseTexture8** interface to manipulate texture resources including cube and volume textures.

The **IDirect3DBaseTexture8** interface assigned to a particular stage for a device is obtained by calling the **IDirect3DDevice8::GetTexture** method.

The **IDirect3DBaseTexture8** interface inherits the following **IDirect3DResource8** methods, which can be organized into these groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The methods of the **IDirect3DBaseTexture8** interface can be organized into the following groups.

Detail	**GetLOD**
	SetLOD
Information	**GetLevelCount**

The **IDirect3DBaseTexture8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DBASETEXTURE8** and **PDIRECT3DBASETEXTURE8** types are defined as pointers to the **IDirect3DBaseTexture8** interface.

```
typedef struct IDirect3DBaseTexture8 *LPDIRECT3DBASETEXTURE8,
*PDIRECT3DBASETEXTURE8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DBaseTexture8::GetLevelCount

Returns the number of texture levels in a multilevel texture.

```
DWORD GetLevelCount();
```

Parameters

None.

Return Values

A **DWORD** value indicating the number of texture levels in a multilevel texture.

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DBaseTexture8**.

- **IDirect3DCubeTexture8**
- **IDirect3DTexture8**
- **IDirect3DVolumeTexture8**

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DBaseTexture8::GetLOD

Returns a value clamped to the maximum level of detail (LOD) set for a managed texture.

```
DWORD GetLOD();
```

Parameters

None.

Return Values

A **DWORD** value, clamped to the maximum LOD value (one less than the total number of levels).

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DBaseTexture8**.

- **IDirect3DCubeTexture8**
- **IDirect3DTexture8**
- **IDirect3DVolumeTexture8**

Remarks

GetLOD is used for LOD control of managed textures. This method returns 0 on nonmanaged textures.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DBaseTexture8::SetLOD

IDirect3DBaseTexture8::SetLOD

Sets the most detailed level of detail (LOD) for a managed texture.

```
DWORD SetLOD(
  DWORD LODNew
);
```

Parameters

LODNew
[in] Most detailed LOD value to set for the mipmap chain.

Return Values

A **DWORD** value, clamped to the maximum LOD value (one less than the total number of levels). Subsequent calls to this method will return the clamped value, not the LOD value that was previously set.

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DBaseTexture8**.

- **IDirect3DCubeTexture8**
- **IDirect3DTexture8**
- **IDirect3DVolumeTexture8**

Remarks

SetLOD is used for LOD control of managed textures. This method returns 0 on nonmanaged textures.

SetLOD communicates to the Microsoft® Direct3D® texture manager the most detailed mipmap in the chain that should be loaded into local video memory. For example, in a five-level mipmap chain, setting *LODNew* to 2 indicates that the texture manager should load only mipmap levels 2 through 4 into local video memory at any given time.

More specifically, if the texture was created with the dimensions of 256×256, setting the most detailed level to 0 indicates that 256×256 is the largest mipmap available, setting the most detailed level to 1 indicates that 128×128 is the largest mipmap available, and so on, up to the most detailed mip level (the smallest texture size) for the chain.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DBaseTexture8::GetLOD

IDirect3DCubeTexture8

Applications use the methods of the **IDirect3DCubeTexture8** interface to manipulate a cube texture resource.

The **IDirect3DCubeTexture8** interface is obtained by calling the **IDirect3DDevice8::CreateCubeTexture** method.

The **IDirect3DCubeTexture8** interface inherits the following **IDirect3DResource8** methods, which can be organized into the following groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The **IDirect3DCubeTexture8** interface inherits the following **IDirect3DBaseTexture8** methods, which can be organized into the following groups.

Detail	**GetLOD**
	SetLOD
Information	**GetLevelCount**

The methods of the **IDirect3DCubeTexture8** interface can be organized into the following groups.

Information	**GetLevelDesc**
Locking Surfaces	**LockRect**
	UnlockRect
Miscellaneous	**AddDirtyRect**

GetCubeMapSurface

The **IDirect3DCubeTexture8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DCUBETEXTURE8** and **PDIRECT3DCUBETEXTURE8** types are defined as pointers to the **IDirect3DCubeTexture8** interface.

```
typedef struct IDirect3DCubeTexture8 *LPDIRECT3DCUBETEXTURE8,
*PDIRECT3DCUBETEXTURE8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CreateCubeTexture

IDirect3DCubeTexture8::AddDirtyRect

Adds a dirty region to a cube texture resource.

```
HRESULT AddDirtyRect(
  D3DCUBEMAP_FACES FaceType,
  CONST RECT* pDirtyRect
);
```

Parameters

FaceType
[in] Member of the **D3DCUBEMAP_FACES** enumerated type, identifying the cube map face.

pDirtyRect
[in] Pointer to a **RECT** structure, specifying the dirty region. Specifying NULL expands the dirty region to cover the entire cube texture.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DCubeTexture8::GetLevelDesc, **IDirect3DCubeTexture8::LockRect**, **IDirect3DCubeTexture8::UnlockRect**

IDirect3DCubeTexture8::GetCubeMapSurface

Retrieves a cube texture map surface.

```
HRESULT GetCubeMapSurface(
  D3DCUBEMAP_FACES FaceType,
  UINT Level,
  IDirect3DSurface8** ppCubeMapSurface
);
```

Parameters

FaceType
[in] Member of the **D3DCUBEMAP_FACES** enumerated type, identifying a cube map face.

Level
[in] Specifies a level of a mipmapped cube texture.

ppCubeMapSurface
[out, retval] Address of a pointer to an **IDirect3DSurface8** interface, representing the returned cube texture map surface.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DCubeTexture8::GetLevelDesc

Retrieves a description of the specified cube texture level

```
HRESULT GetLevelDesc(
  UINT Level,
  D3DSURFACE_DESC* pDesc
);
```

Parameters

Level
[in] Specifies a level of a mipmapped cube texture.

pDesc
[out] Pointer to a **D3DSURFACE_DESC** structure, describing the specified cube texture level.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if one or more of the arguments are invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DCubeTexture8::AddDirtyRect, **IDirect3DCubeTexture8::LockRect**, **IDirect3DCubeTexture8::UnlockRect**

IDirect3DCubeTexture8::LockRect

Locks a rectangle on a cube texture resource.

```
HRESULT LockRect(
  D3DCUBEMAP_FACES FaceType,
  UINT Level,
  D3DLOCKED_RECT* pLockedRect,
  CONST RECT* pRect,
```

DWORD *Flags*
);

Parameters

FaceType
[in] Member of the **D3DCUBEMAP_FACES** enumerated type, identifying a cube map face.

Level
[in] Specifies a level of a mipmapped cube texture.

pLockedRect
[out] Pointer to a **D3DLOCKED_RECT** structure, describing the region to lock.

pRect
[in] Pointer to a rectangle to lock. Specified by a pointer to a **RECT** structure. Specifying NULL for this parameter expands the dirty region to cover the entire cube texture.

Flags
[in] A combination of zero or more locking flags, describing the type of lock to perform.

D3DLOCK_NO_DIRTY_UPDATE
By default, a lock on a resource adds a dirty region to that resource. This flag prevents any changes to the dirty state of the resource. Applications should use this flag when they have additional information about the actual set of regions changed during the lock operation and can then pass this information to **IDirect3DCubeTexture8::AddDirtyRect**.

D3DLOCK_NOSYSLOCK
The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation uses slightly more memory, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long-duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY
The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if one or more of the arguments are invalid.

Remarks

Cube Textures created with POOL_DEFAULT are not-lockable.

The only lockable format for a depth-stencil surface is D3DFMT_D16_LOCKABLE.

A multisample backbuffer cannot be locked.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DCubeTexture8::AddDirtyRect, **IDirect3DCubeTexture8::GetLevelDesc**, **IDirect3DCubeTexture8::UnlockRect**

IDirect3DCubeTexture8::UnlockRect

Unlocks a rectangle on a cube texture resource.

```
HRESULT UnlockRect(
  D3DCUBEMAP_FACES FaceType,
  UINT Level
);
```

Parameters

FaceType
[in] Member of the **D3DCUBEMAP_FACES** enumerated type, identifying a cube map face.

Level
[in] Specifies a level of a mipmapped cube texture.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if one or more of the arguments are invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DCubeTexture8::AddDirtyRect, **IDirect3DCubeTexture8::LockRect**

IDirect3DDevice8

Applications use the methods of the **IDirect3DDevice8** interface to perform DrawPrimitive-based rendering, create resources, work with system-level variables, adjust gamma ramp levels, work with palettes, and create shaders.

The **IDirect3DDevice8** interface is obtained by calling the **IDirect3D8::CreateDevice** method.

The methods of the **IDirect3DDevice8** interface can be organized into the following groups.

Cursors	**SetCursorPosition**
	SetCursorProperties
	ShowCursor
Creation	**CreateAdditionalSwapChain**
	CreateCubeTexture
	CreateDepthStencilSurface
	CreateImageSurface
	CreateIndexBuffer
	CreateRenderTarget
	CreateTexture
	CreateVertexBuffer
	CreateVolumeTexture
Device States	**ApplyStateBlock**
	BeginStateBlock
	CaptureStateBlock
	CreateStateBlock
	DeleteStateBlock
	EndStateBlock
	GetClipStatus
	GetDisplayMode
	GetRenderState

	GetRenderTarget
	GetTransform
	SetClipStatus
	SetRenderState
	SetRenderTarget
	SetTransform
Gamma Ramps	**GetGammaRamp**
	SetGammaRamp
High-Order Primitives	**DeletePatch**
	DrawRectPatch
	DrawTriPatch
Index Data	**GetIndices**
	SetIndices
Information	**GetAvailableTextureMem**
	GetCreationParameters
	GetDeviceCaps
	GetDirect3D
	GetInfo
	GetRasterStatus
Lighting and Materials	**GetLight**
	GetLightEnable
	GetMaterial
	LightEnable
	SetLight
	SetMaterial
Miscellaneous	**CopyRects**
	GetFrontBuffer
	MultiplyTransform
	ProcessVertices
	ResourceManagerDiscardBytes
	TestCooperativeLevel

Palettes	**GetCurrentTexturePalette**
	GetPaletteEntries
	SetCurrentTexturePalette
	SetPaletteEntries
Pixel Shaders	**CreatePixelShader**
	DeletePixelShader
	GetPixelShader
	GetPixelShaderConstant
	GetPixelShaderFunction
	SetPixelShader
	SetPixelShaderConstant
Presentation	**Present**
	Reset
Rendering	**DrawIndexedPrimitive**
	DrawIndexedPrimitiveUP
	DrawPrimitive
	DrawPrimitiveUP
	DrawRectPatch
	DrawTriPatch
Scenes	**BeginScene**
	EndScene
Stream Data	**GetStreamSource**
	SetStreamSource
Surfaces	**GetBackBuffer**
	GetDepthStencilSurface
Textures	**GetTexture**
	GetTextureStageState
	SetTexture
	SetTextureStageState
	UpdateTexture
	ValidateDevice

User-Defined Clip Planes	**GetClipPlane**
	SetClipPlane
Viewports	**Clear**
	GetViewport
	SetViewport
Vertex Shaders	**CreateVertexShader**
	DeleteVertexShader
	GetVertexShader
	GetVertexShaderConstant
	GetVertexShaderDeclaration
	GetVertexShaderFunction
	SetVertexShader
	SetVertexShaderConstant

The **IDirect3DDevice8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DDEVICE8** and **PDIRECT3DDEVICE8** types are defined as pointers to the **IDirect3DDevice8** interface.

```
typedef struct IDirect3DDevice8 *LPDIRECT3DDEVICE8, *PDIRECT3DDEVICE8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3D8::CreateDevice

IDirect3DDevice8::ApplyStateBlock

Applies an existing device-state block to the rendering device.

```
HRESULT ApplyStateBlock(
  DWORD Token
```

);

Parameters

Token
[in] Handle to the device-state block to execute, as returned by a previous call to the **IDirect3DDevice8::EndStateBlock** method.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The value 0xFFFFFFFF is an invalid state block handle.

Applications cannot apply a device-state block while recording another block.

As with all operations that affect the state of the rendering device, it is recommended that you apply state blocks during scene rendering—that is, after calling the **IDirect3DDevice8::BeginScene** method and before calling **IDirect3DDevice8::EndScene**.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::BeginStateBlock, **IDirect3DDevice8::EndStateBlock**, **IDirect3DDevice8::CaptureStateBlock**, **IDirect3DDevice8::CreateStateBlock**, **IDirect3DDevice8::DeleteStateBlock**

IDirect3DDevice8::BeginScene

Begins a scene.

HRESULT BeginScene();

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Applications must call this method before performing any rendering and must call **IDirect3DDevice8::EndScene** when rendering is complete and before calling **BeginScene** again.

If the **BeginScene** method fails, the device was unable to begin the scene, and there is no need to call the **IDirect3DDevice8::EndScene** method. In fact, calls to **EndScene** will fail if the previous call to **BeginScene** failed.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::EndScene

IDirect3DDevice8::BeginStateBlock

Signals Microsoft® Direct3D® to begin recording a device-state block.

```
HRESULT BeginStateBlock();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

Applications can ensure that all recorded states are valid by calling the **IDirect3DDevice8::ValidateDevice** method prior to calling this method.

The following methods can be recorded in a state block, after calling **BeginStateBlock** and before **IDirect3DDevice8::EndStateBlock**.

- **IDirect3DDevice8::LightEnable**
- **IDirect3DDevice8::SetClipPlane**
- **IDirect3DDevice8::SetIndices**
- **IDirect3DDevice8::SetLight**
- **IDirect3DDevice8::SetMaterial**
- **IDirect3DDevice8::SetPixelShader**
- **IDirect3DDevice8::SetPixelShaderConstant**
- **IDirect3DDevice8::SetRenderState**
- **IDirect3DDevice8::SetStreamSource**
- **IDirect3DDevice8::SetTexture**
- **IDirect3DDevice8::SetTextureStageState**
- **IDirect3DDevice8::SetTransform**
- **IDirect3DDevice8::SetViewport**
- **IDirect3DDevice8::SetVertexShader**
- **IDirect3DDevice8::SetVertexShaderConstant**

The ordering of state changes in a state block is not guaranteed. If the same state is specified multiple times in a state block, only the last value is used.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::ApplyStateBlock, **IDirect3DDevice8::EndStateBlock**, **IDirect3DDevice8::CaptureStateBlock**, **IDirect3DDevice8::CreateStateBlock**, **IDirect3DDevice8::DeleteStateBlock**

IDirect3DDevice8::CaptureStateBlock

Updates the values within an existing state block to the values set for the device.

```
HRESULT CaptureStateBlock(
  DWORD Token
);
```

Parameters

Token

[in] Handle to the state block into which the device state is captured.

Return Values

If the method succeeds, the return value is D3D_OK.

If a state block is currently being recorded the method fails and the return value can be D3DERR_INVALIDCALL.

Remarks

The value 0xFFFFFFFF is an invalid state block handle.

This method captures updated values for states within an existing state block. It does not capture the entire state of the device.

CaptureStateBlock will not capture information for lights that are explicitly or implicitly created after the stateblock is created. For example, capturing the current state into a stateblock of type D3DSBT_ALL will not store information for lights that are created post-capture.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::ApplyStateBlock, **IDirect3DDevice8::BeginStateBlock**, **IDirect3DDevice8::CreateStateBlock**, **IDirect3DDevice8::EndStateBlock**, **IDirect3DDevice8::DeleteStateBlock**

IDirect3DDevice8::Clear

Clears the viewport, or a set of rectangles in the viewport, to a specified RGBA color, clears the depth buffer, and erases the stencil buffer.

```
HRESULT Clear(
  DWORD Count,
  CONST D3DRECT* pRects,
  DWORD Flags,
  D3DCOLOR Color,
  float Z,
  DWORD Stencil
);
```

Parameters

Count

[in] Number of rectangles in the array at *pRects*. If you set *pRects* to NULL, this parameter must be set to 0.

pRects

[in] Pointer to an array of **D3DRECT** structures that describe the rectangles to clear. Set a rectangle to the dimensions of the rendering target to clear the entire surface. Each rectangle uses screen coordinates that correspond to points on the render target surface. Coordinates are clipped to the bounds of the viewport rectangle. This parameter can be set to NULL to indicate that the entire viewport rectangle is to be cleared.

Flags

[in] Flags that indicate which surfaces should be cleared. This parameter can be any combination of the following flags, but at least one flag must be used.

D3DCLEAR_STENCIL

Clear the stencil buffer to the value in the *Stencil* parameter.

D3DCLEAR_TARGET

Clear the render target to the color in the *Color* parameter.

D3DCLEAR_ZBUFFER

Clear the depth buffer to the value in the *Z* parameter.

Color

[in] A 32-bit ARGB color value to which the render target surface is cleared.

Z

[in] New z value that this method stores in the depth buffer. This parameter can be in the range from 0.0 through 1.0 (for z-based or w-based depth buffers). A value of 0.0 represents the nearest distance to the viewer, and 1.0 the farthest distance.

Stencil

[in] Integer value to store in each stencil-buffer entry. This parameter can be in the range from 0 through 2^n-1, where *n* is the bit depth of the stencil buffer.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method fails if you specify the D3DCLEAR_ZBUFFER or D3DCLEAR_STENCIL flags when the render target does not have an attached depth buffer. Similarly, if you specify the D3DCLEAR_STENCIL flag when the depth-buffer format does not contain stencil buffer information, this method fails.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::CopyRects

Copies rectangular subsets of pixels from one surface to another.

```
HRESULT CopyRects(
  IDirect3DSurface8* pSourceSurface,
  CONST RECT* pSourceRectsArray,
  UINT cRects,
  IDirect3DSurface8* pDestinationSurface,
  CONST POINT* pDestPointsArray
);
```

Parameters

pSourceSurface
[in] Pointer to an **IDirect3DSurface8** interface, representing the source surface. This parameter must point to a different surface than *pDestinationSurface*.

pSourceRectsArray
[in] Pointer to an array of **RECT** structures, representing the rectangles to be transferred. Each rectangle will be transferred to the destination surface, with its top-left pixel at the position identified by the corresponding element of *pDestPointsArray*. Specifying NULL for this parameter causes the entire surface to be copied.

cRects
[in] Number of **RECT** structures contained in *pSourceRectsArray*.

pDestinationSurface
[in] Pointer to an **IDirect3DSurface8** interface, representing the destination surface. This parameter must point to a different surface than *pSourceSurface*.

pDestPointsArray
[in] Pointer to an array of **POINT** structures, identifying the top-left pixel position of each rectangle contained in *pSourceRectsArray*. If this parameter is NULL, the **RECT**s are copied to the same offset (same top/left location) as the source rectangle.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method does not support stretch, color key, alpha blend, format conversion, or clipping of either source or destination rectangles. Note that this method will fail unless all the source rectangles and their corresponding destination rectangles are completely contained within the source and destination surfaces respectively. The format of the two surfaces must match, but they can have different dimensions.

This method cannot be applied to surfaces whose formats are classified as depth stencil formats.

A **POINT** structure defines the x- and y- coordinates of a point. For more information on **POINT**, see the Microsoft® Platform Software Development Kit (SDK).

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::CreateAdditionalSwapChain

Creates an additional swap chain for rendering multiple views.

```
HRESULT CreateAdditionalSwapChain(
  D3DPRESENT_PARAMETERS* pPresentationParameters,
  IDirect3DSwapChain8** ppSwapChain
);
```

Parameters

pPresentationParameters
[in] Pointer to a **D3DPRESENT_PARAMETERS** structure, representing the presentation parameters for the new swap chain. This value cannot be NULL.

ppSwapChain
[out, retval] Address of a pointer to an **IDirect3DSwapChain8** interface, representing the additional swap chain.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

There is always at least one swap chain (the implicit swap chain) for each device, because Microsoft® Direct3D® for Microsoft DirectX® 8.0 has one swap chain as a property of the device.

Note that any given device can support only one full-screen swap chain.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

Presenting Multiple Views in Windowed Mode

IDirect3DDevice8::CreateCubeTexture

Creates a cube texture resource.

```
HRESULT CreateCubeTexture(
  UINT EdgeLength,
  UINT Levels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  IDirect3DCubeTexture8** ppCubeTexture
);
```

Parameters

EdgeLength
[in] Size of the edges of all the top-level faces of the cube texture. The pixel dimensions of subsequent levels of each face will be the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0 (zero), 1 will be taken instead.

Levels
[in] The number of levels in each face of the cube texture. If this is zero, Microsoft® Direct3D® will generate all cube texture sub-levels down to 1×1 pixels for each face for hardware that supports mipmapped cube textures. Otherwise, it will create one level. Call **IDirect3DBaseTexture8::GetLevelCount** to see the number of levels generated.

Usage

[in] A combination of one or more of the following flags, describing the usage for this resource.

D3DUSAGE_DEPTHSTENCIL
Set to indicate that the surface is to be used as a depth-stencil surface. The resource can be passed to the *pNewDepthStencil* parameter of the **IDirect3DDevice8::SetRenderTarget** method. See Remarks.

D3DUSAGE_RENDERTARGET
Set to indicate that the surface is to be used as a render target. The resource can be passed to the *pRenderTarget* parameter of the **SetRenderTarget** method. See Remarks.

If either D3DUSAGE_RENDERTARGET or D3DUSAGE_DEPTHSTENCIL is specified, the application should check that the device supports these operations by calling **IDirect3D8::CheckDeviceFormat**.

Format
[in] Member of the **D3DFORMAT** enumerated type, describing the format of all levels in all faces of the cube texture.

Pool
[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the cube texture should be placed.

ppCubeTexture
[out, retval] Address of a pointer to an **IDirect3DCubeTexture8** interface, representing the created cube texture resource.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. In order to call **SetRenderTarget** with a cube texture, you must select an individual face using **IDirect3DCubeTexture8::GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

A texture (mipmap) is a collection of successively downsampled (mipmapped) surfaces. On the other hand, a cube texture (created by **CreateCubeTexture**) is a collection of six textures (mipmaps), one for each face. All faces must be present in the cube texture. Also, a cube map surface must be the same pixel size in all three dimensions (x, y, and z).

In DirectX® 8.0, resource usage is enforced. An application that wants to use a resource in a certain operation must specify that operation at resource creation time.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::CreateDepthStencilSurface

Creates a depth-stencil resource.

```
HRESULT CreateDepthStencilSurface(
  UINT Width,
  UINT Height,
  D3DFORMAT Format,
  D3DMULTISAMPLE_TYPE MultiSample,
  IDirect3DSurface8** ppSurface
);
```

Parameters

Width
[in] Width of the depth-stencil surface, in pixels.

Height
[in] Height of the depth-stencil surface, in pixels.

Format
[in] Member of the **D3DFORMAT** enumerated type, describing the format of the depth-stencil surface. This value must be one of the enumerated depth-stencil formats for this device.

MultiSample
[in] Member of the **D3DMULTISAMPLE_TYPE** enumerated type, describing the multisampling buffer type. This value must be one of the allowed multisample types. When this surface is passed to **SetRenderTarget**, its multisample type must be the same as that of the render target.

ppSurface
[out, retval] Address of a pointer to an **IDirect3DSurface8** interface, representing the created depth-stencil surface resource.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

Remarks

The memory class of the depth-stencil buffer is always D3DPOOL_DEFAULT.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CopyRects, **IDirect3DDevice8::SetRenderTarget**

IDirect3DDevice8::CreateImageSurface

Creates an image surface.

```
HRESULT CreateImageSurface(
  UINT Width,
  UINT Height,
  D3DFORMAT Format,
  IDirect3DSurface8** ppSurface
);
```

Parameters

Width
[in] Width of the image surface, in pixels.

Height
[in] Height of the image surface, in pixels.

Format
[in] Member of the **D3DFORMAT** enumerated type, describing the format of the image surface.

ppSurface
[out, retval] Address of a pointer to an **IDirect3DSurface8** interface, representing the created image surface.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

Image surfaces are place holders, they are surfaces that cannot be used in any Microsoft® Direct3D® operations except locking and **IDirect3DDevice8::CopyRects**.

Image surfaces are placed in the D3DPOOL_SYSTEMMEM memory class.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::CreateIndexBuffer

Creates an index buffer.

```
HRESULT CreateIndexBuffer(
  UINT Length,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  IDirect3DIndexBuffer8** ppIndexBuffer
);
```

Parameters

Length
[in] Size of the index buffer, in bytes.

Usage
[in] A combination of one or more of the following flags, describing the usage controls for this resource.

D3DUSAGE_DONOTCLIP
Set to indicate that the index buffer content will never require clipping.

D3DUSAGE_DYNAMIC
Set to indicate when the vertex or index buffer requires dynamic memory usage. This usage is useful for drivers because it enables them to decide where to place the driver. In general, static vertex buffers will be placed in video memory and dynamic vertex buffers will be placed in AGP memory. Note that there is no separate static usage; if you do not specify D3DUSAGE_DYNAMIC the vertex buffer is made static. D3DUSAGE_DYNAMIC is strictly enforced through the

D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE locking flags. As a result, D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE are only valid on vertex and index buffers created with D3DUSAGE_DYNAMIC; they are not valid flags on static vertex buffers.

Note that D3DUSAGE_DYNAMIC cannot be specified on managed vertex and index buffers. For more information, see Managing Resources.

D3DUSAGE_RTPATCHES
Set to indicate when the index buffer is to be used for drawing high-order primitives.

D3DUSAGE_NPATCHES
Set to indicate when the index buffer is to be used for drawing N patches.

D3DUSAGE_POINTS
Set to indicate when the index buffer is to be used for drawing point sprites or indexed point lists.

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the buffer is to be used with software processing.

D3DUSAGE_WRITEONLY
Informs the system that the application writes only to the index buffer. Using this flag enables the driver to select the best memory location for efficient write operations and rendering. Attempts to read from an index buffer that is created with this capability can result in degraded performance.

Format
[in] Member of the **D3DFORMAT** enumerated type, describing the format of the index buffer. The valid settings are the following:

D3DFMT_INDEX16
Indices are 16 bits each.

D3DFMT_INDEX32
Indices are 32 bits each.

See Remarks.

Pool
[in] Member of the **D3DPOOL** enumerated type, describing a valid memory class into which to place the resource.

ppIndexBuffer
[out, retval] Address of a pointer to an **IDirect3DIndexBuffer8** interface, representing the created index buffer resource.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Index buffers are memory resources used to hold indices, they are similar to both surfaces and vertex buffers. The use of index buffers enables Microsoft® Direct3D® to avoid unnecessary data copying and to place the buffer in the optimal memory type for the expected usage.

To use index buffers, create an index buffer, lock it, fill it with indices, unlock it, pass it to **IDirect3DDevice8::SetIndices**, set up the vertices, set up the vertex shader, and call **IDirect3DDevice8::DrawIndexedPrimitive** for rendering.

The **MaxVertexIndex** member of the **D3DCAPS8** structure indicates the types of index buffers that are valid for rendering.

Requirements

Header: Declared in D3d8.h
Import Library: Use D3d8.lib.

See Also

IDirect3DIndexBuffer8::GetDesc

IDirect3DDevice8::CreatePixelShader

Creates a pixel shader.

```
HRESULT CreatePixelShader(
  CONST DWORD* pFunction,
  DWORD* pHandle
);
```

Parameters

pFunction
[in] Pointer to the pixel shader function token array, specifying the blending operations. This value cannot be NULL.

pHandle
[out, retval] Pointer to the returned pixel shader handle.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DeletePixelShader, **D3DXAssembleShader**, **D3DXAssembleShaderFromFileA**, **D3DXAssembleShaderFromFileW**

IDirect3DDevice8::CreateRenderTarget

Creates a render target surface.

```
HRESULT CreateRenderTarget(
  UINT Width,
  UINT Height,
  D3DFORMAT Format,
  D3DMULTISAMPLE_TYPE MultiSample,
  BOOL Lockable,
  IDirect3DSurface8** ppSurface
);
```

Parameters

Width
[in] Width of the render target surface, in pixels.

Height
[in] Height of the render target surface, in pixels.

Format
[in] Member of the **D3DFORMAT** enumerated type, describing the format of the render target.

MultiSample
[in] Member of the **D3DMULTISAMPLE_TYPE** enumerated type, describing the multisampling buffer type. This parameter specifies the antialiasing type for this render target. The type must be the same as that of the depth-stencil buffer when both surfaces are passed to **SetRenderTarget**.

Lockable

[in] Render targets are not lockable unless the application specifies TRUE for *Lockable*. Note that lockable render targets incur a performance cost on some graphics hardware.

ppSurface

[out, retval] Address of a pointer to a **IDirect3DSurface8** interface.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

Render target surfaces are placed in the D3DPOOL_DEFAULT memory class.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::CreateStateBlock

Creates a new state block that contains the values for all device states, vertex-related states, or pixel-related states.

```
HRESULT CreateStateBlock(
  D3DSTATEBLOCKTYPE Type,
  DWORD* pToken
);
```

Parameters

Type

[in] Type of state data that the method should capture. This parameter can be set to a value defined in the **D3DSTATEBLOCKTYPE** enumerated type.

pToken

[out, retval] Pointer to a **DWORD** value to contain the state block handle if the method succeeds.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

The value 0xFFFFFFFF is an invalid state block handle.

Vertex-related device states typically refer to those states that affect how the system processes vertices. Pixel-related states generally refer to device states that affect how the system processes pixel or depth-buffer data during rasterization. Some states are contained in both groups.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::ApplyStateBlock, **IDirect3DDevice8::BeginStateBlock**, **IDirect3DDevice8::CaptureStateBlock**, **IDirect3DDevice8::EndStateBlock**, **IDirect3DDevice8::DeleteStateBlock**

IDirect3DDevice8::CreateTexture

Creates a texture resource.

```
HRESULT CreateTexture(
  UINT Width,
  UINT Height,
  UINT  Levels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  IDirect3DTexture8** ppTexture
);
```

Parameters

Width

[in] Width of the top-level of the texture, in pixels. The pixel dimensions of subsequent levels will be the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0 (zero), 1 will be taken instead.

Height

[in] Height of the top-level of the texture, in pixels. The pixel dimensions of subsequent levels will be the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0 (zero), 1 will be taken instead.

Levels

[in] The number of levels in the texture. If this is zero, Microsoft® Direct3D® will generate all texture sub-levels down to 1×1 pixels for hardware that supports MIP mapped textures. Otherwise, it will create one level. Call **IDirect3DBaseTexture8::GetLevelCount** to see the number of levels generated.

Usage

[in] A combination of one or more of the following flags, describing the usage for this resource.

D3DUSAGE_DEPTHSTENCIL

Set to indicate that the surface is to be used as a depth-stencil surface. The resource can be passed to the *pNewZStencil* parameter of the **IDirect3DDevice8::SetRenderTarget** method.

D3DUSAGE_RENDERTARGET

Set to indicate that the surface is to be used as a render target. The resource can be passed to the *pRenderTarget* parameter of **SetRenderTarget**.

If either D3DUSAGE_RENDERTARGET or D3DUSAGE_DEPTHSTENCIL is specified, the application should check that the device supports these operations by calling **IDirect3D8::CheckDeviceFormat**.

Format

[in] Member of the **D3DFORMAT** enumerated type, describing the format of all levels in the texture.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the texture should be placed.

ppTexture

[out, retval] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture resource.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

In order to call **IDirect3DDevice8::SetRenderTarget** with a texture, you must select a level using **IDirect3DTexture8::GetSurfaceLevel** and pass the resulting surface to **SetRenderTarget**.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::CreateVertexBuffer

Creates a vertex buffer.

```
HRESULT CreateVertexBuffer(
  UINT Length,
  DWORD Usage,
  DWORD FVF,
  D3DPOOL Pool,
  IDirect3DVertexBuffer8** ppVertexBuffer
);
```

Parameters

Length
[in] Size of the vertex buffer, in bytes. For flexible vertex format (FVF) vertex buffers, *Length* must be large enough to contain at least one vertex, but it need not be a multiple of the vertex size. *Length* is not validated for nonFVF buffers. See Remarks.

Usage
[in] A combination of one or more of the following flags, describing the usage controls for this resource.

D3DUSAGE_DONOTCLIP
Set to indicate that the vertex buffer content will never require clipping. When rendering with buffers that have this flag set, the D3DRS_CLIPPING renderstate must be set to false.

D3DUSAGE_DYNAMIC
Set to indicate when the vertex or index buffer requires dynamic memory usage. This usage is useful for drivers because it enables them to decide where to place the driver. In general, static vertex buffers will be placed in video memory and dynamic vertex buffers will be placed in AGP memory. Note that

there is no separate static usage; if you do not specify D3DUSAGE_DYNAMIC the vertex buffer is made static. D3DUSAGE_DYNAMIC is strictly enforced through the D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE locking flags. As a result, D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE are only valid on vertex and index buffers created with D3DUSAGE_DYNAMIC; they are not valid flags on static vertex buffers.

For more information about using dynamic vertex buffers, see Using Dynamic Vertex and Index Buffers.

Note that D3DUSAGE_DYNAMIC cannot be specified on managed vertex and index buffers. For more information, see Managing Resources.

D3DUSAGE_RTPATCHES
Set to indicate when the vertex buffer is to be used for drawing high-order primitives.

D3DUSAGE_NPATCHES
Set to indicate when the vertex buffer is to be used for drawing N patches.

D3DUSAGE_POINTS
Set to indicate when the vertex buffer is to be used for drawing point sprites or indexed point lists.

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the vertex buffer is to be used with software vertex processing.

D3DUSAGE_WRITEONLY
Informs the system that the application writes only to the vertex buffer. Using this flag enables the driver to choose the best memory location for efficient write operations and rendering. Attempts to read from a vertex buffer that is created with this capability will fail.

FVF
[in] Combination of flexible vertex format flags, a usage specifier that describes the vertex format of the vertices in this buffer. When this parameter is set to a valid FVF code, the created vertex buffer is an FVF vertex buffer (see Remarks); otherwise, when this parameter is set to zero, the vertex buffer is a non-FVF vertex buffer.

Pool
[in] Member of the **D3DPOOL** enumerated type, describing a valid memory class into which to place the resource.

ppVertexBuffer
[out,retval] Address of a pointer to an **IDirect3DVertexBuffer8** interface, representing the created vertex buffer resource.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

The **IDirect3DDevice8** interface supports rendering of primitives using vertex data stored in vertex buffer objects. Vertex buffers are created from the **IDirect3DDevice8** interface, and are usable only with the **IDirect3DDevice8** object from which they are created.

When set to a non-zero value, which must be a valid FVF code, the *FVF* parameter indicates that the buffer content is to be characterized by an FVF code. A vertex buffer that is created with an FVF code is referred to as an FVF vertex buffer. For more information, see FVF Vertex Buffers.

The D3DUSAGE_SOFTWAREPROCESSING flag specifies that the vertex buffer is to be used with software vertex processing. For more information, see Device Types and Vertex Processing Requirements.

Non-FVF buffers can be used to interleave data during multipass rendering or multitexture rendering in a single pass. To do this, one buffer contains geometry data and the others contain texture coordinates for each texture to be rendered. When rendering, the buffer containing the geometry data is interleaved with each of the buffers containing the texture coordinates. If FVF buffers were used instead, each of them would need to contain identical geometry data in addition to the texture coordinate data specific to each texture rendered. This would result in either a speed or memory penalty, depending on the strategy used. For more information on texture coordinates, see Understanding Texture Coordinates.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVertexBuffer8::GetDesc, **IDirect3DDevice8::ProcessVertices**

IDirect3DDevice8::CreateVertexShader

Creates a vertex shader and if created successfully sets that shader as the current shader.

HRESULT CreateVertexShader(
CONST DWORD* *pDeclaration*,
CONST DWORD* *pFunction*,

```
  DWORD* pHandle,
  DWORD Usage
);
```

Parameters

pDeclaration

[in, out] Pointer to the vertex shader declaration token array. This parameter defines the inputs to the shader, including how the vertex elements within the input data streams are used by the shader.

pFunction

[in, out] Pointer to the vertex shader function token array. This parameter defines the operations to apply to each vertex. If this parameter is set to NULL, a shader is created for the fixed-function pipeline and the parameter declaration indicated by *pDeclaration* is made current and available to be set in a subsequent call to **IDirect3DDevice8::SetVertexShader** as long as a handle pointer is provided.

If this parameter is not set to NULL, the shader is programmable.

pHandle

[in, out] Pointer to the returned vertex shader handle. This value cannot be set as NULL.

Usage

[in] Usage controls for the vertex shader. The following flag can be set.

D3DUSAGE_SOFTWAREPROCESSING

Set to indicate that the vertex shader is to be used with software vertex processing. The D3DUSAGE_SOFTWAREPROCESSING flag must be set for vertex shaders used when the D3DRS_SOFTWAREVERTEXPROCESSING member of the **D3DRENDERSTATETYPE** enumerated type is TRUE, and removed for vertex shaders used when D3DRS_SOFTWAREVERTEXPROCESSING is FALSE.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

A vertex shader is defined by two token arrays that specify the declaration and function of the shader. The token arrays are composed of single or multiple **DWORD** tokens terminated by a special 0xFFFFFFFF token value.

The shader declaration defines the static external interface of the shader, including binding of stream data to vertex register inputs and values loaded into the shader constant memory. The shader function defines the operation of the shader as an array of instructions that are executed in order for each vertex processed during the time the shader is bound to a device. Shaders created without a function array apply the fixed function vertex processing when that shader is current.

See d3d8types.h for a definition of the macros used to generate the declaration token array.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DeleteVertexShader, **D3DXAssembleShader**, **D3DXAssembleShaderFromFileA**, **D3DXAssembleShaderFromFileW**, **D3DXDeclaratorFromFVF**

IDirect3DDevice8::CreateVolumeTexture

Creates a volume texture resource.

```
HRESULT CreateVolumeTexture(
  UINT Width,
  UINT Height,
  UINT Depth,
  UINT Levels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  IDirect3DVolumeTexture8** ppVolumeTexture
);
```

Parameters

Width

[in] Width of the top-level of the volume texture, in pixels. This value must be a power of two. The pixel dimensions of subsequent levels will be the truncated value of half of the previous level's pixel dimension (independently). Each

dimension clamps at a size of 1 pixel. Thus, if the division by two results in 0 (zero), 1 will be taken instead.

Height

[in] Height of the top-level of the volume texture, in pixels. This value must be a power of two. The pixel dimensions of subsequent levels will be the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0 (zero), 1 will be taken instead.

Depth

[in] Depth of the top-level of the volume texture, in pixels. This value must be a power of two. The pixel dimensions of subsequent levels will be the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0 (zero), 1 will be taken instead.

Levels

[in] The number of levels in the texture. If this is zero, Microsoft® Direct3D® will generate all texture sub-levels down to 1×1×1 pixels for hardware that supports mipmapped volume textures. Otherwise, it will create one level. Call **IDirect3DBaseTexture8::GetLevelCount** to see the number of levels generated.

Usage

[in] Currently not used, set to 0.

Format

[in] Member of the **D3DFORMAT** enumerated type, describing the format of all levels in the volume texture.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the volume texture should be placed.

ppVolumeTexture

[out, retval] Address of a pointer to an **IDirect3DVolumeTexture8** interface, representing the created volume texture resource.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::DeletePixelShader

Deletes the pixel shader referred to by the specified handle.

```
HRESULT DeletePixelShader(
  DWORD Handle
);
```

Parameters

Handle
[in] Pixel shader handle, identifying the pixel shader to be deleted from its internal entry.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CreatePixelShader

IDirect3DDevice8::DeleteStateBlock

Deletes a previously recorded device-state block.

```
HRESULT DeleteStateBlock(
  DWORD Token
);
```

Parameters

Token
[in] Handle to the device-state block to delete, as returned by a previous call to the **IDirect3DDevice8::EndStateBlock** method.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The value 0xFFFFFFFF is an invalid state block handle.

Applications cannot delete a device-state block while another is being recorded

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::ApplyStateBlock, **IDirect3DDevice8::BeginStateBlock**, **IDirect3DDevice8::CaptureStateBlock**, **IDirect3DDevice8::CreateStateBlock**, **IDirect3DDevice8::EndStateBlock**

IDirect3DDevice8::DeletePatch

Frees a cached high-order patch.

```
HRESULT DeletePatch(
  UINT Handle
);
```

Parameters

Handle
[in] Handle of the cached high-order patch to delete.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DrawRectPatch, **IDirect3DDevice8::DrawTriPatch**, Drawing Patches

IDirect3DDevice8::DeleteVertexShader

Deletes the vertex shader referred to by the specified handle and frees up the associated resources.

```
HRESULT DeleteVertexShader(
  DWORD Handle
);
```

Parameters

Handle
[in] Vertex shader handle, identifying the vertex shader to be deleted.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CreateVertexShader

IDirect3DDevice8::DrawIndexedPrimitive

Renders the specified geometric primitive, based on indexing into an array of vertices.

```
HRESULT DrawIndexedPrimitive(
  D3DPRIMITIVETYPE Type,
  UINT MinIndex,
  UINT NumVertices,
  UINT StartIndex,
  UINT PrimitiveCount
);
```

Parameters

Type
[in] Member of the **D3DPRIMITIVETYPE** enumerated type, describing the type of primitive to render. See Remarks.

MinIndex
[in] Minimum vertex index for vertices used during this call.

NumVertices
[in] The number of vertices used during this call starting from *BaseVertexIndex* + *MinIndex*

StartIndex
[in] Location in the index array to start reading vertices.

PrimitiveCount
[in] Number of primitives to render. The number of vertices used is a function of the primitive count and the primitive type. The maximum number of primitives allowed is determined by checking the *MaxPrimitiveCount* member of the **D3DCAPS8** structure.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method draws indexed primitives from the current set of data input streams.

MinIndex and all the indices in the index stream are relative to the *BaseVertexIndex*, set during the **IDirect3DDevice8::SetIndices** call.

The *minIndex* and *NumVertices* parameters specify the range of vertex indices used for each **DrawIndexedPrimitive** call. These are used to optimize vertex processing of indexed primitives by processing a sequential range of vertices prior to indexing into these vertices. It is invalid for any indices used during this call to reference any vertices outside of this range.

DrawIndexedPrimitive fails if no index array is set.

The D3DPT_POINTLIST member of the **D3DPRIMITIVETYPE** enumerated type is not supported and should not be specified for *Type*.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DrawPrimitive, **IDirect3DDevice8::SetStreamSource**, Rendering Primitives

IDirect3DDevice8::DrawIndexedPrimitiveUP

Renders the specified geometric primitive with data specified by a user memory pointer.

```
HRESULT DrawIndexedPrimitiveUP(
  D3DPRIMITIVETYPE PrimitiveType,
  UINT MinIndex,
  UINT NumVertices,
  UINT PrimitiveCount,
  CONST void* pIndexData,
  D3DFORMAT IndexDataFormat,
  CONST void* pVertexStreamZeroData,
  UINT VertexStreamZeroStride
);
```

Parameters

PrimitiveType
[in] Member of the **D3DPRIMITIVETYPE** enumerated type, describing the type of primitive to render.

MinIndex
[in] Minimum vertex index, relative to zero (the start of *pIndexData*), for vertices used during this call.

NumVertices
[in] Number of vertices used during this call; starting from *MinIndex*.

PrimitiveCount
[in] Number of primitives to render. The number of indices used is a function of the primitive count and the primitive type. The maximum number of primitives allowed is determined by checking the *MaxPrimitiveCount* member of the **D3DCAPS8** structure.

pIndexData
[in] User memory pointer to the index data.

IndexDataFormat
[in] Member of the **D3DFORMAT** enumerated type, describing the format of the index data. The valid settings are the following:

D3DFMT_INDEX16
Indices are 16 bits each.

D3DFMT_INDEX32

Indices are 32 bits each.

pVertexStreamZeroData
[in] User memory pointer to vertex data to use for vertex stream zero.

VertexStreamZeroStride
[in] Stride between data for each vertex, in bytes.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method is intended for use in applications that are unable to store their vertex data in vertex buffers. This method supports only a single vertex stream. The effect of this call is to use the provided vertex data pointer and stride for vertex stream zero. It is invalid to have the declaration of the current vertex shader refer to vertex streams other than stream zero.

Following any **DrawIndexedPrimitiveUP** call, the stream zero settings, referenced by **IDirect3DDevice8::GetStreamSource**, are set to NULL. Also, the index buffer setting for **IDirect3DDevice8::SetIndices** is set to NULL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DrawPrimitiveUP, **IDirect3DDevice8::SetStreamSource**, Rendering Primitives

IDirect3DDevice8::DrawPrimitive

Renders a sequence of nonindexed, geometric primitives of the specified type from the current set of data input streams.

```
HRESULT DrawPrimitive(
  D3DPRIMITIVETYPE PrimitiveType,
  UINT StartVertex,
  UINT PrimitiveCount
);
```

Parameters

PrimitiveType
[in] Member of the **D3DPRIMITIVETYPE** enumerated type, describing the type of primitive to render.

StartVertex
[in] Index of the first vertex to load. Beginning at *StartVertex* the correct number of vertices will be read out of the vertex buffer.

PrimitiveCount
[in] Number of primitives to render. The maximum number of primitives allowed is determined by checking the *MaxPrimitiveCount* member of the **D3DCAPS8** structure. *PrimitiveCount* is the number of primitives as determined by the primitive type. If it is a line list, each primitive has two vertices. If it is a triangle list, each primitive has three vertices.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

DrawPrimitive should not be called with a single triangle at a time.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DrawIndexedPrimitive, **IDirect3DDevice8::SetStreamSource**, Rendering Primitives

IDirect3DDevice8::DrawPrimitiveUP

Renders data specified by a user memory pointer as a sequence of geometric primitives of the specified type.

```
HRESULT DrawPrimitiveUP(
  D3DPRIMITIVETYPE PrimitiveType,
  UINT PrimitiveCount,
  CONST void* pVertexStreamZeroData,
  UINT VertexStreamZeroStride
);
```

Parameters

PrimitiveType
[in] Member of the **D3DPRIMITIVETYPE** enumerated type, describing the type of primitive to render.

PrimitiveCount
[in] Number of primitives to render. The maximum number of primitives allowed is determined by checking the *MaxPrimitiveCount* member of the **D3DCAPS8** structure.

pVertexStreamZeroData
[in] User memory pointer to vertex data to use for vertex stream zero.

VertexStreamZeroStride
[in] Stride between data for each vertex, in bytes.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method is intended for use in applications that are unable to store their vertex data in vertex buffers. This method supports only a single vertex stream. The effect of this call is to use the provided vertex data pointer and stride for vertex stream zero. It is invalid to have the declaration of the current vertex shader refer to vertex streams other than stream zero.

Following any **DrawPrimitiveUP** call, the stream zero settings, referenced by **IDirect3DDevice8::GetStreamSource**, are set to NULL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DrawIndexedPrimitiveUP,
IDirect3DDevice8::SetStreamSource, Rendering Primitives

IDirect3DDevice8::DrawRectPatch

Draws a rectangular high-order patch using the currently set streams.

```
HRESULT DrawRectPatch(
  UINT Handle,
  CONST float* pNumSegs,
```

```
  CONST D3DRECTPATCH_INFO* pRectPatchInfo
);
```

Parameters

Handle
[in] Handle to the rectangular high-order patch to draw.

pNumSegs
[in, out] Pointer to a floating-point value identifying the number of segments that each edge of the high-order primitive should be divided into when tessellated.

pRectPatchInfo
[in, out] Pointer to a **D3DRECTPATCH_INFO** structure, describing the rectangular high-order patch to draw.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DeletePatch, Drawing Patches

IDirect3DDevice8::DrawTriPatch

Draws a triangular high-order patch using the currently set streams.

```
HRESULT DrawTriPatch(
  UINT Handle,
  CONST float* pNumSegs,
  CONST D3DTRIPATCH_INFO* pTriPatchInfo
);
```

Parameters

Handle
[in] Handle to the triangular high-order patch to draw.

pNumSegs
[in, out] Pointer to a floating-point value identifying the number of segments that each edge of the high-order primitive should be divided into when tessellated.

pTriPatchInfo
[in, out] Pointer to a **D3DTRIPATCH_INFO** structure, describing the triangular high-order patch to draw.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DeletePatch, Drawing Patches

IDirect3DDevice8::EndScene

Ends a scene that was begun by calling the **IDirect3DDevice8::BeginScene** method.

```
HRESULT EndScene();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

When this method succeeds, the scene has been rendered and the device surface holds the rendered scene.

When scene rendering begins successfully, you must call this method before you can call the **IDirect3DDevice8::BeginScene** method to start rendering another scene. If a prior call to **BeginScene** method fails, the scene did not begin and this method should not be called.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::BeginScene

IDirect3DDevice8::EndStateBlock

Signals Microsoft® Direct3D® to stop recording a device-state block and retrieve a handle to the state block.

```
HRESULT EndStateBlock(
  DWORD* pToken
);
```

Parameters

pToken
[out, retval] Pointer to a variable to fill with the handle to the completed device-state block. This value is used with the **IDirect3DDevice8::ApplyStateBlock** and **IDirect3DDevice8::DeleteStateBlock** methods.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

Remarks

The value 0xFFFFFFFF is an invalid state block handle.

See Also

IDirect3DDevice8::ApplyStateBlock, **IDirect3DDevice8::BeginStateBlock**, **IDirect3DDevice8::CaptureStateBlock**, **IDirect3DDevice8::CreateStateBlock**, **IDirect3DDevice8::DeleteStateBlock**

IDirect3DDevice8::GetAvailableTextureMem

Returns an estimate of the amount of available texture memory.

```
UINT GetAvailableTextureMem();
```

Parameters

None.

Return Values

The function returns an estimate of the available texture memory.

Remarks

The returned value is rounded to the nearest MB. This is done to reflect the fact that video memory estimates are never precise due to alignment and other issues that affect consumption by certain resources. Applications can use this value to make gross estimates of memory availability to make large-scale resource decisions such as how many levels of a mipmap to attempt to allocate, but applications cannot use this value to make small-scale decisions such as if there is enough memory left to allocate another resource.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::GetBackBuffer

Retrieves a back buffer from the device's swap chain.

```
HRESULT GetBackBuffer(
  UINT BackBuffer,
  D3DBACKBUFFER_TYPE Type,
  IDirect3DSurface8** ppBackBuffer
);
```

Parameters

BackBuffer
[in] Index of the back buffer object to return. Back buffers are numbered from 0 to the total number of back buffers - 1. A value of 0 returns the first back buffer, not the front buffer. The front buffer is not accessible through this method.

Type
[in] Stereo view is not supported in DirectX 8.0, so the only valid value for this parameter is D3DBACKBUFFER_TYPE_MONO.

ppBackBuffer
[out, retval] Address of a pointer to an **IDirect3DSurface8** interface, representing the returned back buffer surface.

Return Values

If the method succeeds, the return value is D3D_OK.

If *BackBuffer* exceeds or equals the total number of back buffers, then the function fails and returns D3DERR_INVALIDCALL.

Note
Calling this method will increase the internal reference count on the **IDirect3DSurface8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DSurface8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::GetClipPlane

Retrieves the coefficients of a user-defined clipping plane for the device.

```
HRESULT GetClipPlane(
  DWORD Index,
  float* pPlane
);
```

Parameters

Index
[in] Index of the clipping plane for which the plane equation coefficients are retrieved.

pPlane
[out] Pointer to a four-element array of values that represent the coefficients of the clipping plane in the form of the general plane equation. See Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is D3DERR_INVALIDCALL. This error indicates that the value in *Index* exceeds the maximum clipping plane index supported by the device, or that the array at *pPlane* is not large enough to contain four floating-point values.

Remarks

The coefficients that this method reports take the form of the general plane equation. If the values in the array at *pPlane* were labeled A, B, C, and D in the order that they appear in the array, they would fit into the general plane equation so that Ax + By + Cz + Dw = 0. A point with homogeneous coordinates (x, y, z, w) is visible in the half space of the plane if Ax + By + Cz + Dw >= 0. Points that exist on or behind the clipping plane are clipped from the scene.

The plane equation used by this method exists in world space and is set by a previous call to the **IDirect3DDevice8::SetClipPlane** method.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetClipPlane

IDirect3DDevice8::GetClipStatus

Retrieves the clip status.

```
HRESULT GetClipStatus(
  D3DCLIPSTATUS8* pClipStatus
);
```

Parameters

pClipStatus
[out] Pointer to a **D3DCLIPSTATUS8** structure that describes the clip status.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if the argument is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetClipStatus

IDirect3DDevice8::GetCreationParameters

Retrieves the creation parameters of the device.

```
HRESULT GetCreationParameters(
  D3DDEVICE_CREATION_PARAMETERS* pParameters
);
```

Parameters

pParameters
[out] Pointer to a **D3DDEVICE_CREATION_PARAMETERS** structure, describing the creation parameters of the device.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if the argument is invalid.

Remarks

You can query the **AdapterOrdinal** member of the returned **D3DDEVICE_CREATION_PARAMETERS** structure to retrieve the ordinal of the adapter represented by this device.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::GetCurrentTexturePalette

Retrieves the current texture palette.

```
HRESULT GetCurrentTexturePalette(
```

```
  UINT* pPaletteNumber
);
```

Parameters

pPaletteNumber
[out, retval] Pointer to a returned value that identifies the current texture palette.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetCurrentTexturePalette, Texture Palettes

IDirect3DDevice8::GetDepthStencilSurface

Retrieves the depth-stencil surface owned by the Direct3DDevice object.

```
HRESULT GetDepthStencilSurface(
  IDirect3DSurface8** ppZStencilSurface
);
```

Parameters

ppZStencilSurface
[out, retval] Address of a pointer to an **IDirect3DSurface8** interface, representing the returned depth-stencil surface.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count on the **IDirect3DSurface8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DSurface8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetRenderTarget, **IDirect3DDevice8::CopyRects**

IDirect3DDevice8::GetDeviceCaps

Retrieves the capabilities of the rendering device.

```
HRESULT GetDeviceCaps(
  D3DCAPS8* pCaps
);
```

Parameters

pCaps
[out] Pointer to a **D3DCAPS8** structure, describing the returned device.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

GetDeviceCaps retrieves the software vertex pipeline capabilities when the device is being used in software vertex processing mode. Software vertex processing mode is selected when a device has been created with D3DCREATE_SOFTWAREVERTEXPROCESSING, or when a device has been created with D3DCREATE_MIXEDVERTEXPROCESSING and D3DRS_SOFTWAREVERTEXPROCESSING is set to TRUE.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::GetDirect3D

Returns an interface to the instance of the Microsoft® Direct3D® object that created the device.

```
HRESULT GetDirect3D(
  IDirect3D8** ppD3D8
);
```

Parameters

ppD3D8
[out, retval] Address of a pointer to an **IDirect3D8** interface, representing the interface of the Direct3D object that created the device.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Calling **GetDirect3D** increases the reference count on the interface, so applications should call **IUnknown::Release** through this pointer when finished with the interface.

Note

Calling this method will increase the internal reference count on the **IDirect3D8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3D8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::GetDisplayMode

Retrieves the display mode's spatial resolution, color resolution, and refresh frequency.

```
HRESULT GetDisplayMode(
  D3DDISPLAYMODE* pMode
);
```

Parameters

pMode

[out] Pointer to a **D3DDISPLAYMODE** structure containing data about the display mode of the adapter. As opposed to the display mode of the device, which may not be active if the device does not own full-screen mode.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::GetFrontBuffer

Generates a copy of the device's front buffer and places that copy in a system memory buffer provided by the application.

```
HRESULT GetFrontBuffer(
  IDirect3DSurface8* pDestSurface
);
```

Parameters

pDestSurface

[in] Pointer to an **IDirect3DSurface8** interface that will receive a copy of the contents of the front buffer. The data is returned in successive rows with no intervening space, starting from the vertically highest row on the device's output to the lowest.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

If the method fails, the return value can be one of the following values.

D3DERR_DEVICELOST

D3DERR_INVALIDCALL

Remarks

The buffer pointed to by *pDestSurface* will be filled with a representation of the front buffer, converted to the standard 32bpp format, D3DFMT_A8R8G8B8.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::GetGammaRamp

Retrieves the gamma correction ramp for the swap chain.

```
void GetGammaRamp(
  D3DGAMMARAMP* pRamp
);
```

Parameters

pRamp
[in, out] Pointer to an application-supplied **D3DGAMMARAMP** structure to fill with the gamma correction ramp.

Return Values

None.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DSwapChain8, **IDirect3DDevice8::SetGammaRamp**

IDirect3DDevice8::GetIndices

Retrieves index data.

```
HRESULT GetIndices(
  IDirect3DIndexBuffer8** ppIndexData,
  UINT* pBaseVertexIndex
);
```

Parameters

ppIndexData
[out] Address of a pointer to an **IDirect3DIndexBuffer8** interface, representing the returned index data.

pBaseVertexIndex
[out] Pointer to a **UINT** value, holding the returned base value for vertex indices. This value is added to all indices prior to referencing vertex data, defining a starting position in the vertex streams.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The values in the index list are used to index into a vertex list when creating geometry to render.

The value returned in the *pBaseVertexIndex* parameter specifies the base value for indices. This base value is added to all indices prior to referencing into the vertex data streams, the result of which is to set a starting position in the vertex data streams. The base vertex index enables multiple indexed primitives to be packed into a single set of vertex data without requiring the indices to be recomputed based on where the corresponding primitive is placed in the vertex data.

Note

Calling this method will increase the internal reference count on the **IDirect3DIndexBuffer8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DIndexBuffer8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetIndices

IDirect3DDevice8::GetInfo

Retrieves information about the rendering device.

HRESULT GetInfo(
DWORD *DevInfoID*,

VOID* *pDevInfoStruct*,
DWORD *DevInfoStructSize*
);

Parameters

DevInfoID
[in] Value used to identify what information will be returned in *pDevInfoStruct*.

pDevInfoStruct
[in, out] Pointer to a structure that receives the specified device information if the call succeeds.

DevInfoStructSize
[in] Size of the structure at *pDevInfoStruct*, in bytes.

Return Values

If the method succeeds, the return value is D3D_OK. This method returns S_FALSE on retail builds of Microsoft® DirectX® (see Remarks).

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Information returned by this method can pertains to the underlying device driver. This method makes it possible for drivers to declare specific information types, and corresponding structures, that are not documented in this SDK.

This method executes synchronously and can negatively impact an application's performance when it executes slowly. Do not call this method during scene rendering (between calls to **IDirect3DDevice8::BeginScene** and **IDirect3DDevice8::EndScene**).

This method is intended to be used for performance tracking and debugging during product development (on the debug version of DirectX). The method can succeed, returning S_FALSE, without retrieving device data. This occurs when the retail version of the DirectX runtime is installed on the host system.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::GetLight

Retrieves a set of lighting properties that this device uses.

HRESULT GetLight(
DWORD *Index*,

```
  D3DLIGHT8* pLight
);
```

Parameters

Index
[in] Zero-based index of the lighting property set to retrieve.

pLight
[out] Pointer to a **D3DLIGHT8** structure that is filled with the retrieved lighting-parameter set.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL if the *pLight* parameter is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetLight, **IDirect3DDevice8::GetLightEnable**, **IDirect3DDevice8::LightEnable**

IDirect3DDevice8::GetLightEnable

Retrieves the activity status—enabled or disabled—for a set of lighting parameters within a device.

```
HRESULT GetLightEnable(
  DWORD Index,
  BOOL* pEnable
);
```

Parameters

Index
[in] Zero-based index of the set of lighting parameters that are the target of this method.

pEnable
[out, retval] Pointer to a variable to fill with the status of the specified lighting parameters. After the call, a nonzero value at this address indicates that the

specified lighting parameters are enabled; a value of 0 indicates that they are disabled.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetLight, **IDirect3DDevice8::LightEnable**, **IDirect3DDevice8::SetLight**

IDirect3DDevice8::GetMaterial

Retrieves the current material properties for the device.

```
HRESULT GetMaterial(
  D3DMATERIAL8*pMaterial
);
```

Parameters

pMaterial
[out] Pointer to a **D3DMATERIAL8** structure to fill with the currently set material properties.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL if the *pMaterial* parameter is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetMaterial

IDirect3DDevice8::GetPaletteEntries

Retrieves palette entries.

```
HRESULT GetPaletteEntries(
  UINT PaletteNumber,
  PALETTEENTRY* pEntries
);
```

Parameters

PaletteNumber
[in] An ordinal value identifying the particular palette to retrieve.

pEntries
[in, out] Pointer to a **PALETTEENTRY** structure, representing the returned palette entries. See Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

For more information on **PALETTEENTRY** see the Microsoft® Platform Software Development Kit (SDK). Note that as of DirectX 8.0 the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetCurrentTexturePalette,
IDirect3DDevice8::SetCurrentTexturePalette,
IDirect3DDevice8::SetPaletteEntries, Texture Palettes

IDirect3DDevice8::GetPixelShader

Retrieves the currently set pixel shader.

```
HRESULT GetPixelShader(
  DWORD* pHandle
```

);

Parameters

pHandle
[out, retval] Pointer to a pixel shader handle, representing the returned the pixel shader.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetPixelShader

IDirect3DDevice8::GetPixelShaderConstant

Retrieves the values in the pixel constant array.

```
HRESULT GetPixelShaderConstant(
  DWORD Register,
  void* pConstantData,
  DWORD ConstantCount
);
```

Parameters

Register
[in] Register address at which to start retrieving data from the pixel constant array.

pConstantData
[in, out] Pointer to the data block to hold the retrieved values from the pixel constant array. The size of the data block is (*ConstantCount* * 4 * sizeof(float)).

ConstantCount
[in] Number of constants to retrieve from the pixel constant array. Each constant is comprised of four floating-point values.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This is the method used to retrieve the values in the constant registers of the pixel shader assembler.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetPixelShaderConstant

IDirect3DDevice8::GetPixelShaderFunction

Retrieves the pixel shader function.

```
HRESULT GetPixelShaderFunction(
  DWORD Handle,
  void* pData,
  DWORD* pSizeOfData
);
```

Parameters

Handle
[in] Handle to the referred to pixel shader.

pData
[out] Pointer to a previously allocated buffer to be filled with the code associated with the requested pixel shader handle, if the call succeeds. The application calling this method in responsible for allocating and releasing this buffer.

pSizeOfData
[in, out] Pointer to a **DWORD** value, indicating the size of the buffer at *pData*, in bytes. If this value is less than the actual size of the data (such as 0) the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA. If the buffer is NULL, then this parameter is filled with the required size and D3D_OK is returned.

Return Values

If the method succeeds, the return value is D3D_OK.

The return value can be D3DERR_INVALIDCALL if *Handle* is an invalid handle to a pixel shader.

Remarks

The pixel shader function specifies blending operations.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetPixelShader

IDirect3DDevice8::GetRasterStatus

Returns information describing the raster of the monitor on which the swap chain is presented.

```
HRESULT GetRasterStatus(
  D3DRASTER_STATUS* pRasterStatus
);
```

Parameters

pRasterStatus
[out] Pointer to a **D3DRASTER_STATUS** structure filled with information about the position or other status of the raster on the monitor driven by this adapter.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if *pRasterStatus* is invalid or if the device does not support reading the current scan line. To determine whether or not the device supports reading the scan line, check for the D3DCAPS_READ_SCANLINE flag in the **Caps** member of **D3DCAPS8**.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

D3DRASTER_STATUS

IDirect3DDevice8::GetRenderState

Retrieves a render-state value for a device.

```
HRESULT GetRenderState(
  D3DRENDERSTATETYPE State,
  DWORD* pValue
);
```

Parameters

State
[in] Device state variable that is being queried. This parameter can be any member of the **D3DRENDERSTATETYPE** enumerated type.

pValue
[out, retval] Pointer to a variable that receives the value of the queried render state variable when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL if one of the arguments is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetRenderState

IDirect3DDevice8::GetRenderTarget

Retrieves a pointer to the Microsoft® Direct3D® surface that is being used as a render target.

```
HRESULT GetRenderTarget(
```

IDirect3DSurface8** *ppRenderTarget*
);

Parameters

ppRenderTarget
[out, retval] Address of a pointer to an **IDirect3DSurface8** interface, representing the returned render target surface for this device.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL if one of the arguments is invalid.

Note
Calling this method will increase the internal reference count on the **IDirect3DSurface8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DSurface8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetRenderTarget

IDirect3DDevice8::GetStreamSource

Retrieves a vertex buffer bound to the specified data stream.

HRESULT GetStreamSource(
UINT *StreamNumber*
IDirect3DVertexBuffer8** *ppStreamData,*
UINT* *pStride*
);

Parameters

StreamNumber
[in] Specifies the data stream, in the range from 0 to the maximum number of streams - 1.

ppStreamData

[in, out] Address of a pointer to an **IDirect3DVertexBuffer8** interface, representing the returned vertex buffer bound to the specified data stream.

pStride

[in, out] Pointer to a returned stride of the component, in bytes. See Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A stream is defined as a uniform array of component data, where each component consists of one or more elements representing a single entity such as position, normal, color, and so on.

Note

Calling this method will increase the internal reference count on the **IDirect3DVertexBuffer8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DVertexBuffer8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetStreamSource

IDirect3DDevice8::GetTexture

Retrieves a texture assigned to a stage for a device.

```
HRESULT GetTexture(
  DWORD Stage,
  IDirect3DBaseTexture8** ppTexture
);
```

Parameters

Stage

[in] Stage identifier of the texture to retrieve. Stage identifiers are zero-based. Devices can have up to eight set textures, so the maximum value allowed for *Stage* is 7.

ppTexture

[out, retval] Address of a pointer to an **IDirect3DBaseTexture8** interface, representing the returned texture.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count on the **IDirect3DTexture8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DTexture8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetTexture, **IDirect3DDevice8::GetTextureStageState**, **IDirect3DDevice8::SetTextureStageState**

IDirect3DDevice8::GetTextureStageState

Retrieves a state value for an assigned texture.

```
HRESULT GetTextureStageState(
  DWORD Stage,
  D3DTEXTURESTAGESTATETYPE Type,
  DWORD* pValue
);
```

Parameters

Stage

[in] Stage identifier of the texture for which the state is retrieved. Stage identifiers are zero-based. Devices can have up to eight set textures, so the maximum value allowed for *Stage* is 7.

Type

[in] Texture state to retrieve. This parameter can be any member of the **D3DTEXTURESTAGESTATETYPE** enumerated type.

pValue

[out, retval] Pointer a variable to fill with the retrieved state value. The meaning of the retrieved value is determined by the *Type* parameter.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetTextureStageState, **IDirect3DDevice8::GetTexture**, **IDirect3DDevice8::SetTexture**

IDirect3DDevice8::GetTransform

Retrieves a matrix describing a transformation state.

```
HRESULT GetTransform(
  D3DTRANSFORMSTATETYPE State,
  D3DMATRIX* pMatrix
);
```

Parameters

State
[in] Device state variable that is being modified. This parameter can be any member of the **D3DTRANSFORMSTATETYPE** enumerated type, or the **D3DTS_WORLDMATRIX** macro.

pMatrix
[out] Pointer to a **D3DMATRIX** structure, describing the returned transformation state.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL if one of the arguments is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetTransform

IDirect3DDevice8::GetVertexShader

Retrieves the currently set vertex shader.

```
HRESULT GetVertexShader(
  DWORD* pHandle
);
```

Parameters

pHandle
[out, retval] Pointer to a vertex shader handle, representing the returned vertex shader.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if *pHandle* is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetVertexShader

IDirect3DDevice8::GetVertexShaderConstant

Retrieves the values in the vertex constant array.

```
HRESULT GetVertexShaderConstant(
  DWORD Register,
  void* pConstantData,
  DWORD ConstantCount
);
```

Parameters

Register

[in] Register address at which to start retrieving data from the vertex constant array.

pConstantData

[out] Pointer to the data block to hold the retrieved values from the vertex constant array. The size of the data block is (*ConstantCount* * 4 * sizeof(float)).

ConstantCount

[in] Number of constants to retrieve from the vertex constant array. Each constant is comprised of four floating-point values.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This is the method used to retrieve the values in the constant registers of the vertex shader assembler.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetVertexShaderConstant

IDirect3DDevice8::GetVertexShaderDeclaration

Retrieves the vertex shader declaration token array.

```
HRESULT GetVertexShaderDeclaration(
  DWORD Handle,
  void* pData,
  DWORD* pSizeOfData
);
```

Parameters

Handle

[in] Handle to the referred to vertex shader.

pData

[in] Pointer to a previously allocated buffer to be filled with the declaration associated with the requested vertex shader handle, if the call succeeds. The

application calling this method is responsible for allocating and releasing this buffer.

pSizeOfData

[in, out] Pointer to a **DWORD** value, indicating the size of the buffer at *pData*, in bytes. If this value is less than the actual size of the data (such as 0) the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA. If the buffer is NULL, then this parameter is filled with the required size and D3D_OK is returned.

Return Values

If the method succeeds, the return value is D3D_OK.

The return value can be D3DERR_INVALIDCALL if *Handle* is an invalid handle to a vertex shader.

Remarks

The vertex shader declaration token array defines the inputs to the shader, including how the vertex elements within the input data streams are used by the shader.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetVertexShader

IDirect3DDevice8::GetVertexShaderFunction

Retrieves the vertex shader function.

```
HRESULT GetVertexShaderFunction(
  DWORD Handle,
  void* pData,
  DWORD* pSizeOfData
);
```

Parameters

Handle

[in] Handle to the referred to vertex shader.

pData

[out] Pointer to a previously allocated buffer to be filled with the code associated with the requested vertex shader handle, if the call succeeds. The application calling this method is responsible for allocating and releasing this buffer.

pSizeOfData

[in, out] Pointer to a **DWORD** value, indicating the size of the buffer at *pData*, in bytes. If this value is less than the actual size of the data (such as 0) the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA. If the buffer is NULL, then this parameter is filled with the required size and D3D_OK is returned.

Return Values

If the method succeeds, the return value is D3D_OK.

The return value can be D3DERR_INVALIDCALL if *Handle* is an invalid handle to a vertex shader.

Remarks

The vertex shader function defines the operation to apply to each vertex.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetVertexShader

IDirect3DDevice8::GetViewport

Retrieves the viewport parameters currently set for the device.

```
HRESULT GetViewport(
  D3DVIEWPORT8* pViewport
);
```

Parameters

pViewport

[out] Pointer to a **D3DVIEWPORT8** structure, representing the returned viewport parameters.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if the *pViewport* parameter is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetViewport

IDirect3DDevice8::LightEnable

Enables or disables a set of lighting parameters within a device.

```
HRESULT LightEnable(
  DWORD LightIndex,
  BOOL bEnable
);
```

Parameters

LightIndex
[in] Zero-based index of the set of lighting parameters that are the target of this method.

bEnable
[in] Value that indicates if the set of lighting parameters are being enabled or disabled. Set this parameter to TRUE to enable lighting with the parameters at the specified index, or FALSE to disable it.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

If a value for *LightIndex* is outside the range of the light property sets assigned within the device, the **LightEnable** method creates a light source represented by a **D3DLIGHT8** structure with the following properties and sets its enabled state to the value specified in *bEnable*.

Member	Default
Type	D3DLIGHT_DIRECTIONAL
Diffuse	(R:1, G:1, B:1, A:0)

Specular	(R:0, G:0, B:0, A:0)
Ambient	(R:0, G:0, B:0, A:0)
Position	(0, 0, 0)
Direction	(0, 0, 1)
Range	0
Falloff	0
Attenuation0	0
Attenuation1	0
Attenuation2	0
Theta	0
Phi	0

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetLight, **IDirect3DDevice8::GetLightEnable**, **IDirect3DDevice8::SetLight**

IDirect3DDevice8::MultiplyTransform

Multiplies a device's world, view, or projection matrices by a specified matrix.

```
HRESULT MultiplyTransform(
  D3DTRANSFORMSTATETYPE State,
  CONST D3DMATRIX* pMatrix
);
```

Parameters

State
[in] Member of the **D3DTRANSFORMSTATETYPE** enumerated type, or the **D3DTS_WORLDMATRIX** macro that identifies which device matrix is to be modified. The most common setting, D3DTS_WORLDMATRIX(0), modifies the world matrix, but you can specify that the method modify the view or projection matrices, if needed.

pMatrix
[in] Pointer to a **D3DMATRIX** structure that modifies the current transformation.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL if one of the arguments is invalid.

Remarks

The multiplication order is *pMatrix* times *State*.

An application might use the **MultiplyTransform** method to work with hierarchies of transformations. For example, the geometry and transformations describing an arm might be arranged in the following hierarchy.

```
shoulder_transformation
   upper_arm geometry
   elbow transformation
      lower_arm geometry
      wrist transformation
         hand geometry
```

An application might use the following series of calls to render this hierarchy. Not all the parameters are shown in this pseudocode.

```
IDirect3DDevice8::SetTransform(D3DTS_WORLDMATRIX(0),
   shoulder_transform)
IDirect3DDevice8::DrawPrimitive(upper_arm)
IDirect3DDevice8::MultiplyTransform(D3DTS_WORLDMATRIX(0),
   elbow_transform)
IDirect3DDevice8::DrawPrimitive(lower_arm)
IDirect3DDevice8::MultiplyTransform(D3DTS_WORLDMATRIX(0),
   wrist_transform)
IDirect3DDevice8::DrawPrimitive(hand)
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DrawPrimitive, **IDirect3DDevice8::SetTransform**, **D3DTS_WORLD**, **D3DTS_WORLD*n***, **D3DTS_WORLDMATRIX**

IDirect3DDevice8::Present

Presents the contents of the next in the sequence of back buffers owned by the device.

HRESULT Present(

```
  CONST RECT* pSourceRect,
  CONST RECT* pDestRect,
  HWND hDestWindowOverride,
  CONST RGNDATA* pDirtyRegion
);
```

Parameters

pSourceRect

[in] Pointer to a value that must be NULL unless the swap chain was created with D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC. *pSourceRect*is a pointer to a **RECT** structure containing the source rectangle. If NULL, the entire source surface is presented. If the rectangle exceeds the source surface, the rectangle is clipped to the source surface.

pDestRect

[in] Pointer to a value that must be NULL unless the swap chain was created with D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC. *PDestRect* is a pointer to a **RECT** structure containing the destination rectangle, in window client coordinates. If NULL, the entire client area is filled. If the rectangle exceeds the destination client area, the rectangle is clipped to the destination client area.

hDestWindowOverride

[in] Pointer to a destination window whose client area is taken as the target for this presentation. If this value is NULL, then the **hWndDeviceWindow** member of **D3DPRESENT_PARAMETERS** is taken.

pDirtyRegion

[in] This parameter is not used and should be set to NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_DEVICELOST

Remarks

If necessary, a stretch operation is applied to transfer the pixels within the source rectangle to the destination rectangle in the client area of the target window.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::Reset

IDirect3DDevice8::ProcessVertices

Applies the vertex processing defined by the vertex shader to the set of input data streams, generating a single stream of interleaved vertex data to the destination vertex buffer.

```
HRESULT ProcessVertices(
  UINT SrcStartIndex,
  UINT DestIndex,
  UINT VertexCount,
  IDirect3DVertexBuffer8* pDestBuffer,
  DWORD Flags
);
```

Parameters

SrcStartIndex
[in] Index of first vertex to be loaded.

DestIndex
[in] Index of first vertex in the destination vertex buffer into which the results are placed.

VertexCount
[in] Number of vertices to process.

pDestBuffer
[in] Pointer to an **IDirect3DVertexBuffer8** interface, the destination vertex buffer representing the stream of interleaved vertex data.

Flags
[in] Processing options. Set this parameter to 0 for default processing. Set to D3DPV_DONOTCOPYDATA to prevent the system from copying vertex data not affected by the vertex operation into the destination buffer.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The destination vertex buffer, *pDestBuffer*, must be created with a nonzero *FVF* parameter. The FVF code specified during the call to the

IDirect3DDevice8::CreateVertexBuffer method specifies the vertex elements present in the destination vertex buffer.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

Device Types and Vertex Processing Requirements

IDirect3DDevice8::Reset

Resets the type, size, and format of the swap chain.

```
HRESULT Reset(
  D3DPRESENT_PARAMETERS* pPresentationParameters
);
```

Parameters

pPresentationParameters
[in] Pointer to a **D3DPRESENT_PARAMETERS** structure, describing the new presentation parameters. This value cannot be NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

If a call to **Reset** fails, the device will be placed in the "lost" state (as indicated by a return value of D3DERR_DEVICELOST from a call to **IDirect3DDevice::TestCooperativeLevel**) unless it is already in the "not reset" state (as indicated by a return value of D3DERR_DEVICENOTRESET from a call to **IDirect3DDevice8::TestCooperativeLevel**). Refer to **IDirect3DDevice8::TestCooperativeLevel** and Lost Devices for further information concerning the use of **Reset** in the context of lost devices.

Calling **Reset** causes all texture memory surfaces to be lost, managed textures to be flushed from video memory, and all state information to be lost. Before calling the **Reset** method for a device, an application should release any explicit render targets, depth stencil surfaces, additional swap chains and D3DPOOL_DEFAULT resources associated with the device.

The different types of swap chains are full-screen or windowed. If the new swap chain is full-screen, the adapter will be placed in the display mode that matches the new size.

DirectX 8.0 applications can expect messages to be sent to them during this call (for example, before this call is returned); applications should take precautions not to call into Direct3D at this time. In addition, when **Reset** fails, the only valid methods that can be called are **Reset**, **IDirect3DDevice8::TestCooperativeLevel**, and the various **Release** member functions. Calling any other method may result in an exception.

A call to **Reset** will fail if called on a different thread than that used to create the device being reset.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

D3DSWAPEFFECT, **D3DPRESENT_PARAMETERS**, **IDirect3DDevice8::Present**

IDirect3DDevice8::ResourceManagerDiscardBytes

Invokes the resource manager to free memory.

```
HRESULT ResourceManagerDiscardBytes(
  DWORD Bytes
);
```

Parameters

Bytes
[in] The number of target bytes to discard. If zero, then the resource manager should discard all bytes.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The resource manager frees up memory allocations until the number of target bytes is reached. The resource manager follows a least-recently-used (LRU) priority technique.

Note that the number of bytes freed are not guaranteed to be contiguous. Therefore you might not be able to create a resource that takes the number of bytes specified in *Bytes* after calling this method.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::SetClipPlane

Sets the coefficients of a user-defined clipping plane for the device.

```
HRESULT SetClipPlane(
  DWORD Index,
  CONST float* pPlane
);
```

Parameters

Index
[in] Index of the clipping plane for which the plane equation coefficients are to be set.

pPlane
[in] Pointer to an address of a four-element array of values that represent the clipping plane coefficients to be set, in the form of the general plane equation. See Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is D3DERR_INVALIDCALL. This error indicates that the value in *Index* exceeds the maximum clipping plane index supported by the device or that the array at *pPlane* is not large enough to contain four floating-point values.

Remarks

The coefficients that this method sets take the form of the general plane equation. If the values in the array at *pPlane* were labeled A, B, C, and D in the order that they appear in the array, they would fit into the general plane equation so that Ax + By + Cz + Dw = 0. A point with homogeneous coordinates (x, y, z, w) is visible in the half space of the plane if Ax + By + Cz + Dw >= 0. Points that exist on or behind the clipping plane are clipped from the scene.

When the fixed function pipeline is used the plane equations are assumed to be in world space. When the programmable pipeline is used the plane equations are assumed to be in the clipping space (the same space as output vertices).

This method does not enable the clipping plane equation being set. To enable a clipping plane, set the corresponding bit in the **DWORD** value applied to the D3DRS_CLIPPLANEENABLE render state.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetClipPlane

IDirect3DDevice8::SetClipStatus

Sets the clip status.

```
HRESULT SetClipStatus(
  CONST D3DCLIPSTATUS8* pClipStatus
);
```

Parameters

pClipStatus
[in] Pointer to a **D3DCLIPSTATUS8** structure, describing the clip status settings to be set.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL if one of the arguments is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetClipStatus

IDirect3DDevice8::SetCurrentTexturePalette

Sets the current texture palette.

```
HRESULT SetCurrentTexturePalette(
  UINT PaletteNumber
);
```

Parameters

PaletteNumber
[in] Value that specifies the texture palette to set as the current texture palette.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A single logical palette is associated with the device, and is shared by all texture stages.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetCurrentTexturePalette, Texture Palettes

IDirect3DDevice8::SetCursorPosition

Sets the cursor position and update options.

```
void SetCursorPosition(
  UINT XScreenSpace,
  UINT YScreenSpace,
  DWORD Flags
);
```

Parameters

XScreenSpace
[in] The new X-position of the cursor in screen-space coordinates. See Remarks.

YScreenSpace
[in] The new Y-position of the cursor in screen-space coordinates. See Remarks.

Flags
[in] Specifies the update options for the cursor. Currently, only one flag is defined.

D3DCURSOR_IMMEDIATE_UPDATE
Update cursor at the refresh rate.

If this flag is specified, the system guarantees that the cursor will be updated at a minimum of half the display refresh rate, but never more frequently than the display refresh rate. Otherwise, the method delays cursor updates until the next **IDirect3DDevice8::Present** call. This default behavior usually results in better performance than if the flag had been set. However, applications should set this flag if the rate of calls to **Present** is high enough that users would notice a significant delay in cursor motion.

Return Values

None.

Remarks

When running in full-screen mode, screen-space coordinates are the back-buffer coordinates appropriately scaled to the current display mode. When running in windowed mode, screen-space coordinates are the desktop coordinates. The cursor image is drawn at the specified position minus the hotspot offset specified by the **IDirect3DDevice8::SetCursorProperties** method.

If the cursor has been hidden by **IDirect3DDevice8::ShowCursor**, then the cursor is not drawn.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetCursorProperties, **IDirect3DDevice8::ShowCursor**

IDirect3DDevice8::SetCursorProperties

Sets properties for the cursor.

```
HRESULT SetCursorProperties(
  UINT XHotSpot,
  UINT YHotSpot,
  IDirect3DSurface8* pCursorBitmap
);
```

Parameters

XHotSpot
[in] X-coordinate offset into the cursor, in pixels from the top-left corner, that is considered the center. When the cursor is given a new position, the image is drawn at an offset from this new position determined by subtracting the hot spot coordinates from the position.

YHotSpot
[in] Y-coordinate offset into the cursor, in pixels from the top-left corner, that is considered the center. When the cursor is given a new position, the image is drawn at an offset from this new position determined by subtracting the hot spot coordinates from the position.

pCursorBitmap
[in] Pointer to an **IDirect3DSurface8** interface. This parameter must point to an 8888 ARGB surface (format D3DFORMAT_A8R8G8B8). The contents of this surface will be copied and potentially format-converted into an internal buffer from which the cursor is displayed. The dimensions of this surface must be less than the dimensions of the display mode, and must be a power of two in each direction, although not necessarily the same power of two. The alpha channel must be either 0.0 or 1.0.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Microsoft® Direct3D® cursor functions use either GDI cursor or software emulation, depending on the hardware. Users usually want to respond to a WM_SETCURSOR message. For example, the users might want to write the message handler like this:

```
case WM_SETCURSOR:
    // Turn off window cursor
    SetCursor( NULL );
    m_pd3dDevice->ShowCursor( TRUE );
    return TRUE; // prevent Windows from setting cursor to window class cursor
break;
```

Or users might want to call the **IDirect3DDevice8::SetCursorProperties** method if they want to change the cursor. See the sample code in the Microsoft DirectX® Graphics C/C++ Samples for more details.

The application can determine what hardware support is available for cursors by examining appropriate members of the **D3DCAPS8** structure. Typically, hardware supports only 32x32 cursors. Additionally, when windowed, the system may support only 32x32 cursors. In this case, **SetCursorProperties** still succeeds, but the cursor may be reduced to that size—the hot spot is scaled appropriately.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetCursorPosition, **IDirect3DDevice8::ShowCursor**, **D3DCAPS8**

IDirect3DDevice8::SetGammaRamp

Sets the gamma correction ramp for the implicit swap chain.

```
void SetGammaRamp(
  DWORD Flags,
  CONST D3DGAMMARAMP* pRamp
);
```

Parameters

Flags

[in] Indicates whether correction should be applied. Gamma correction results in a more consistent display, but can incur processing overhead and should not be used frequently. Short-duration effects such as flashing the whole screen red should not be calibrated, but long-duration gamma changes should be calibrated. One of the following values can be set.

D3DSGR_CALIBRATE

If a gamma calibrator is installed, the ramp will be modified before being sent to the device to account for the system and monitor response curves.

If a calibrator is not installed, the ramp will be passed directly to the device.

D3DSGR_NO_CALIBRATION
No gamma correction is applied. The supplied gamma table is transferred directly to the device.

pRamp
[in] Pointer to a **D3DGAMMARAMP** structure, representing the gamma correction ramp to be set for the implicit swap chain.

Return Values

None.

Remarks

There is always at least one swap chain (the implicit swap chain) for each device, because Microsoft® Direct3D® for Microsoft DirectX® 8.0 has one swap chain as a property of the device.

Because the gamma ramp is a property of the swap chain, the gamma ramp may be applied when the swap chain is windowed.

The gamma ramp takes effect immediately. No wait for VSYNC is performed.

If the device does not support gamma ramps in the swap chain's current presentation mode (full-screen or windowed), no error return is given. Applications can check the D3DCAPS2_FULLSCREENGAMMA and D3DCAPS2_CANCALIBRATEGAMMA capability bits in the **Caps2** member of the **D3DCAPS8** structure to determine the capabilities of the device and whether a calibrator is installed.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DSwapChain8, **IDirect3DDevice8::GetGammaRamp**

IDirect3DDevice8::SetIndices

Sets index data.

```
HRESULT SetIndices(
  IDirect3DIndexBuffer8* pIndexData
  UINT BaseVertexIndex
);
```

Parameters

pIndexData
[in] Pointer an **IDirect3DIndexBuffer8** interface, representing the index data to be set.

BaseVertexIndex
[in] Base value for vertex indices. This value is added to all indices prior to referencing vertex data, defining a starting position in the vertex streams.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The **SetIndices** method sets the current index array to an index buffer. The single set of indices is used to index all streams.

The *BaseVertexIndex* parameter specifies the base value for indices. This base value is added to all indices prior to referencing into the vertex data streams, the result of which is to set a starting position in the vertex data streams. The base vertex index enables multiple indexed primitives to be packed into a single set of vertex data without requiring the indices to be recomputed based on where the corresponding primitive is placed in the vertex data.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetIndices

IDirect3DDevice8::SetLight

Assigns a set of lighting properties for this device.

```
HRESULT SetLight(
  DWORD Index,
  CONST D3DLIGHT8* pLight
);
```

Parameters

Index

[in] Zero-based index of the set of lighting properties to set. If a set of lighting properties exists at this index, it is overwritten by the new properties specified in *pLight*.

pLight

[in] Pointer to a **D3DLIGHT8** structure, containing the lighting-parameters to set.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetLight, **IDirect3DDevice8::GetLightEnable**, **IDirect3DDevice8::LightEnable**

IDirect3DDevice8::SetMaterial

Sets the material properties for the device.

```
HRESULT SetMaterial(
  CONST D3DMATERIAL8* pMaterial
);
```

Parameters

pMaterial

[in] Pointer to a **D3DMATERIAL8** structure, describing the material properties to set.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL if the *pMaterial* parameter is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetMaterial

IDirect3DDevice8::SetPaletteEntries

Sets palette entries.

```
HRESULT SetPaletteEntries(
  UINT PaletteNumber,
  CONST PALETTEENTRY* pEntries
);
```

Parameters

PaletteNumber
[in] An ordinal value identifying the particular palette upon which the operation is to be performed.

pEntries
[in] Pointer to a **PALETTEENTRY** structure, representing the palette entries to set. The number of **PALETTEENTRY** structures pointed to by *pEntries* is assumed to be 256. See Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A single logical palette is associated with the device, and is shared by all texture stages.

For more information on **PALETTEENTRY** see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work as it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetCurrentTexturePalette,
IDirect3DDevice8::GetPaletteEntries,
IDirect3DDevice8::SetCurrentTexturePalette, Texture Palettes

IDirect3DDevice8::SetPixelShader

Sets the current pixel shader to a previously created pixel shader.

```
HRESULT SetPixelShader(
  DWORD Handle
);
```

Parameters

Handle

[in] Handle to the pixel shader, specifying the pixel shader to set. The value for this parameter can be the handle returned by **IDirect3DDevice8::CreatePixelShader**.

If the handle is NULL, the legacy pixel pipeline is used.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetPixelShader

IDirect3DDevice8::SetPixelShaderConstant

Sets the values in the pixel constant array.

```
HRESULT SetPixelShaderConstant(
  DWORD Register,
  CONST void* pConstantData,
  DWORD ConstantCount
);
```

Parameters

Register
[in] Register address at which to start loading data into the pixel constant array.

pConstantData
[in] Pointer to the data block holding the values to load into the pixel constant array. The size of the data block is (*ConstantCount* * 4 * sizeof(float)).

ConstantCount
[in] Number of constants to load into the pixel constant array. Each constant is comprised of four floating-point values.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This is the method used to load the constant registers of the pixel shader assembler.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetPixelShaderConstant

IDirect3DDevice8::SetRenderState

Sets a single device render-state parameter.

```
HRESULT SetRenderState(
  D3DRENDERSTATETYPE State,
  DWORD Value
);
```

Parameters

State
[in] Device state variable that is being modified. This parameter can be any member of the **D3DRENDERSTATETYPE** enumerated type.

Value

[in] New value for the device render state to be set. The meaning of this parameter is dependent on the value specified for *State*. For example, if *State* were D3DRS_SHADEMODE, the second parameter would be one member of the **D3DSHADEMODE** enumerated type.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if one of the arguments is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetRenderState, **IDirect3DDevice8::SetTransform**

IDirect3DDevice8::SetRenderTarget

Sets a new color buffer, depth buffer, or both for the device.

```
HRESULT SetRenderTarget(
  IDirect3DSurface8* pRenderTarget,
  IDirect3DSurface8* pNewZStencil
);
```

Parameters

pRenderTarget

[in] Pointer to a new color buffer. If NULL, the existing color buffer is retained. If this parameter is not NULL, the reference count on the new render target is incremented. Devices always have to be associated with a color buffer.

The new render target surface must have at least D3DUSAGE_RENDERTARGET specified.

pNewZStencil

[in] Pointer to a new depth-stencil buffer. If there is an existing depth-stencil buffer, it is released. If this parameter is not NULL, the reference count on the new depth-stencil buffer surface is incremented. Applications can change the render target without changing the depth buffer by passing in the *ppZStencilSurface* parameter of **IDirect3DDevice8::GetDepthStencilSurface**.

The new depth-stencil surface must have at least D3DUSAGE_DEPTHSTENCIL and D3DPOOL_DEFAULT specified.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if *pRenderTarget* or *pNewZStencil* are not NULL and invalid, or if the new depth buffer is smaller than the new or retained color buffer.

Remarks

Once a color buffer and a depth-stencil surface have been associated with the same device by this method, they are said to be paired.

The device will call **AddRef** on each non-NULL surface passed to **SetRenderTarget**. After that the device calls **Release** on the previously set color buffer.

The previous depth-stencil surface's contents persist after a call to **SetRenderTarget** to disassociate the previous depth-stencil surface from the device. If the surface is re-associated with the device, then the contents of the surface will be unchanged, providing the color buffer to which the new depth-stencil surface is being paired is the same size and format as the color buffer to which the depth-stencil surface was most recently paired.

Calling this method resets the current viewport of the device to the size of the current render target. This is done regardless of whether the z-buffer is the only one that is being changed.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CreateDepthStencilSurface,
IDirect3DDevice8::GetDepthStencilSurface

IDirect3DDevice8::SetStreamSource

Binds a vertex buffer to a device data stream.

```
HRESULT SetStreamSource(
  UINT StreamNumber,
  IDirect3DVertexBuffer8* pStreamData,
  UINT Stride
);
```

Parameters

StreamNumber
[in] Specifies the data stream, in the range from 0 to the maximum number of streams - 1.

pStreamData
[in] Pointer to an **IDirect3DVertexBuffer8** interface, representing the vertex buffer to bind to the specified data stream.

Stride
[in] Stride of the component, in bytes. See Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method increments the reference count of the stream being assigned. When the texture is no longer needed, set the texture at the appropriate stage to NULL. If you fail to do this, the surface is not released, resulting in a memory leak.

When a flexible vertex format (FVF) vertex shader is used, the stream vertex stride in **SetStreamSource** must match the vertex size, computed from the FVF. When a declaration is used, the stream vertex stride in **SetStreamSource** should be greater than or equal to the stream size computed from the declaration.

The **SetStreamSource** method binds a vertex buffer to a device data stream. For details, see Setting the Stream Source.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::DrawIndexedPrimitive, **IDirect3DDevice8::DrawIndexedPrimitiveUP**, **IDirect3DDevice8::DrawPrimitive**, **IDirect3DDevice8::DrawPrimitiveUP**, **IDirect3DDevice8::GetStreamSource**

IDirect3DDevice8::SetTexture

Assigns a texture to a stage for a device.

HRESULT SetTexture(
DWORD *Stage*,

IDirect3DBaseTexture8* *pTexture*
);

Parameters

Stage
[in] Stage identifier to which the texture is set. Stage identifiers are zero-based. Devices can have up to eight set textures, so the maximum value allowed for *Stage* is 7.

pTexture
[in] Pointer to an **IDirect3DBaseTexture8** interface, representing the texture being set. For complex textures, such as mipmaps and cube textures, this parameter must point to the top-level surface.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method increments the reference count of the texture surface being assigned and decrements the reference count of the previously selected texture if there is one. When the texture is no longer needed, set the texture at the appropriate stage to NULL. Failure to do this results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetTexture, **IDirect3DDevice8::GetTextureStageState**, **IDirect3DDevice8::SetTextureStageState**

IDirect3DDevice8::SetTextureStageState

Sets the state value for the currently assigned texture.

HRESULT SetTextureStageState(
DWORD *Stage*,
D3DTEXTURESTAGESTATETYPE *Type*,
DWORD *Value*
);

Parameters

Stage
[in] Stage identifier of the texture for which the state value is set. Stage identifiers are zero-based. Devices can have up to eight set textures, so the maximum value allowed for *Stage* is 7.

Type
[in] Texture state to set. This parameter can be any member of the **D3DTEXTURESTAGESTATETYPE** enumerated type.

Value
[in] State value to set. The meaning of this value is determined by the *Type* parameter.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetTextureStageState, **IDirect3DDevice8::GetTexture**, **IDirect3DDevice8::SetTexture**

IDirect3DDevice8::SetTransform

Sets a single device transformation-related state.

```
HRESULT SetTransform(
  D3DTRANSFORMSTATETYPE State,
  CONST D3DMATRIX* pMatrix
);
```

Parameters

State
[in] Device-state variable that is being modified. This parameter can be any member of the **D3DTRANSFORMSTATETYPE** enumerated type, or the **D3DTS_WORLDMATRIX** macro.

pMatrix
[in] Pointer to a **D3DMATRIX** structure that modifies the current transformation.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if one of the arguments is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetTransform, **IDirect3DDevice8::SetRenderState**, **D3DTS_WORLD**, **D3DTS_WORLD*n***, **D3DTS_WORLDMATRIX**

IDirect3DDevice8::SetVertexShader

Sets the current vertex shader to a previously created vertex shader or to a flexible vertex format (FVF) fixed function shader.

```
HRESULT SetVertexShader(
  DWORD Handle
);
```

Parameters

Handle
[in] Handle to a vertex shader, specifying the vertex shader to set. The value for this parameter can be the handle returned by **IDirect3DDevice8::CreateVertexShader**, or an FVF code. An FVF code is a combination of flexible vertex format flags.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The effect of using an FVF code in place of the handle is to enable a fixed-function vertex shader, with an implicit declaration that matches the FVF code contents read from stream zero. Only stream zero is referenced when an FVF-specified shader is bound to the device.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetVertexShader

IDirect3DDevice8::SetVertexShaderConstant

Sets values in the vertex constant array.

```
HRESULT SetVertexShaderConstant(
  DWORD Register,
  CONST void* pConstantData,
  DWORD  ConstantCount
);
```

Parameters

Register
[in] Register address at which to start loading data into the vertex constant array.

pConstantData
[in] Pointer to the data block holding the values to load into the vertex constant array. The size of the data block is (*ConstantCount* * 4 * sizeof(float)).

ConstantCount
[in] Number of constants to load into the vertex constant array. Each constant is comprised of four floating-point values.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This is the method used to load the constant registers of the vertex shader assembler. When loading transformation matrices, the application should transpose them row/column and load them into consecutive constant registers.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetVertexShaderConstant

IDirect3DDevice8::SetViewport

Sets the viewport parameters for the device.

```
HRESULT SetViewport(
  CONST D3DVIEWPORT8* pViewport
);
```

Parameters

pViewport
[in] Pointer to a **D3DVIEWPORT8** structure, specifying the viewport parameters to set.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL if the *pViewport* parameter is invalid.

If the viewport parameters described by the **D3DVIEWPORT8** structure describe a region that cannot exist within the render target surface, the method fails, returning D3DERR_INVALIDCALL.

Remarks

Any call to SetViewPort between a pair of calls to **IDirect3DDevice8::BeginScene** and **IDirect3DDevice8::EndScene** must be made before any geometry is drawn.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetViewport

IDirect3DDevice8::ShowCursor

Displays or hides the cursor.

```
BOOL ShowCursor(
  BOOL bShow
);
```

Parameters

bShow

[in] If *bShow* is TRUE, the cursor is shown. If *bShow* is FALSE, the cursor is hidden.

Return Values

Value indicating whether the cursor was previously visible. TRUE if the cursor was previously visible, or FALSE if the cursor was not previously visible.

Remarks

Microsoft® Direct3D® cursor functions use either GDI cursor or software emulation, depending on the hardware. Users usually want to respond to a WM_SETCURSOR message. For example, the users might want to write the message handler like this:

```
case WM_SETCURSOR:
    // Turn off window cursor
    SetCursor( NULL );
    m_pd3dDevice->ShowCursor( TRUE );
    return TRUE; // prevent Windows from setting cursor to window class cursor
break;
```

Or users might want to call the **IDirect3DDevice8::SetCursorProperties** method if they want to change the cursor. See the code in the Microsoft DirectX® Graphics C/C++ Samples for more detail.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::SetCursorPosition, **IDirect3DDevice8::SetCursorProperties**

IDirect3DDevice8::TestCooperativeLevel

Reports the current cooperative-level status of the Microsoft® Direct3D® device for a windowed or full-screen application.

HRESULT TestCooperativeLevel();

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK, indicating that the device is operational and the calling application can continue.

If the method fails, the return value can be one of the following values (see Remarks).

D3DERR_DEVICELOST

D3DERR_DEVICENOTRESET

Remarks

If the device is lost but cannot be restored at the current time, **IDirect3DDevice8::TestCooperativeLevel** returns the D3DERR_DEVICELOST return code. This would be the case, for example, when a full-screen device has lost focus. If an application detects a lost device, it should pause and periodically call **TestCooperativeLevel** until it receives a return value of D3DERR_DEVICENOTRESET. The application may then attempt to reset the device by calling **IDirect3DDevice8::Reset** and, if this succeeds, restore the necessary resources and resume normal operation. Note that **IDirect3DDevice8::Present** will return D3DERR_DEVICELOST if the device is either "lost" or "not reset".

A call to **TestCooperative Level** will fail if called on a different thread than that used to create the device being reset.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DDevice8::UpdateTexture

Updates the dirty portions of a texture.

HRESULT UpdateTexture(

IDirect3DBaseTexture8* *pSourceTexture*,
IDirect3DBaseTexture8* *pDestinationTexture*
);

Parameters

pSourceTexture
[in] Pointer to an **IDirect3DBaseTexture8** interface, representing the source texture. The source texture must be in system memory (D3DPOOL_SYSTEMMEM).

pDestinationTexture
[in] Pointer to an **IDirect3DBaseTexture8** interface, representing the destination texture. The destination texture must be in the D3DPOOL_DEFAULT memory pool.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

You can dirty a portion of a texture by locking it, or by calling one of the following methods.

- **IDirect3DCubeTexture8::AddDirtyRect**
- **IDirect3DTexture8::AddDirtyRect**
- **IDirect3DVolumeTexture8::AddDirtyBox**
- **IDirect3DDevice8::CopyRects**

UpdateTexture retrieves the dirty portions of the texture by calculating what has been accumulated since the last update operation.

This method fails if the textures are of different types, if their bottom-level buffers are of different sizes, and also if their matching levels do not match. For example, consider a six level source texture with the following dimensions.

32×16, 16×8, 8×4, 4×2, 2×1, 1×1

This six level source texture could be the source for the following one level destination.

1×1

For the following two level destination.

2×1, 1×1

Or, for the following three level destination.

4×2, 2×1, 1×1

In addition, this method will fail if the textures are of different formats. If the destination texture has fewer levels than the source, only the matching levels are copied.

If the source texture has dirty regions, the copy may be optimized by restricting the copy to only those regions. It is not guaranteed that only those bytes marked dirty will be copied.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3D8::CreateDevice

IDirect3DDevice8::ValidateDevice

Reports the device's ability to render the current texture-blending operations and arguments in a single pass.

```
HRESULT ValidateDevice(
  DWORD* pNumPasses
);
```

Parameters

pNumPasses
[out, retval] Pointer to a **DWORD** value to fill with the number of rendering passes needed to complete the desired effect through multipass rendering.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_CONFLICTINGTEXTUREFILTER

D3DERR_CONFLICTINGTEXTUREPALETTE

D3DERR_TOOMANYOPERATIONS

D3DERR_UNSUPPORTEDALPHAARG

D3DERR_UNSUPPORTEDALPHAOPERATION

D3DERR_UNSUPPORTEDCOLORARG

D3DERR_UNSUPPORTEDCOLOROPERATION

D3DERR_UNSUPPORTEDFACTORVALUE

D3DERR_UNSUPPORTEDTEXTUREFILTER

D3DERR_WRONGTEXTUREFORMAT

Remarks

The **ValidateDevice** method should be used to validate scenarios only when other capabilities are deficient. For example, in a multistage texturing scenario, you could query the **MaxTextureBlendStages** and **MaxSimultaneousTextures** members of a **D3DCAPS8** structure to determine if multistage texturing is possible on the device.

Current hardware does not necessarily implement all possible combinations of operations and arguments. You can determine whether a particular blending operation can be performed with given arguments by setting the desired blending operation, and then calling the **ValidateDevice** method.

The **ValidateDevice** method uses the current render states, textures, and texture-stage states to perform validation at the time of the call. Changes to these factors after the call invalidate the previous result, and the method must be called again before rendering a scene.

Using diffuse iterated values, either as an argument or as an operation (D3DTA_DIFFUSE or D3DTOP_BLENDDIFFUSEALPHA) is rarely supported on current hardware. Most hardware can introduce iterated color data only at the last texture operation stage.

Try to specify the texture (D3DTA_TEXTURE) for each stage as the first argument, rather than the second argument.

Many cards do not support use of diffuse or scalar values at arbitrary texture stages. Often, these are available only at the first or last texture-blending stage.

Many cards do not have a blending unit associated with the first texture that is capable of more than replicating alpha to color channels or inverting the input. Therefore, your application might need to use only the second texture stage, if possible. On such hardware, the first unit is presumed to be in its default state, which has the first color argument set to D3DTA_TEXTURE with the D3DTOP_SELECTARG1 operation.

Operations on the output alpha that are more intricate than, or substantially different from, the color operations are less likely to be supported.

Some hardware does not support simultaneous use of D3DTA_TFACTOR and D3DTA_DIFFUSE.

Many cards do not support simultaneous use of multiple textures and mipmapped trilinear filtering. If trilinear filtering has been requested for a texture involved in multitexture blending operations and validation fails, turn off trilinear filtering and revalidate. In this case, you might want to perform multipass rendering instead.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::GetTextureStageState,
IDirect3DDevice8::SetTextureStageState

IDirect3DIndexBuffer8

Applications use the methods of the **IDirect3DIndexBuffer8** interface to manipulate an index buffer resource.

The **IDirect3DIndexBuffer8** interface is obtained by calling the **IDirect3DDevice8::CreateIndexBuffer** method.

The **IDirect3DIndexBuffer8** interface inherits the following **IDirect3DResource8** methods, which can be organized into these groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The methods of the **IDirect3DIndexBuffer8** interface can be organized into the following groups.

Information	**GetDesc**
Locking	**Lock**
	Unlock

The **IDirect3DIndexBuffer8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DINDEXBUFFER8** and **PDIRECT3DINDEXBUFFER8** types are defined as pointers to the **IDirect3DIndexBuffer8** interface.

```
typedef struct IDirect3DIndexBuffer8 *LPDIRECT3DINDEXBUFFER8,
*PDIRECT3DINDEXBUFFER8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CreateIndexBuffer

IDirect3DIndexBuffer8::GetDesc

Retrieves a description of the index buffer resource.

```
HRESULT GetDesc(
  D3DINDEXBUFFER_DESC* pDesc
);
```

Parameters

pDesc
[out] Pointer to a **D3DINDEXBUFFER_DESC** structure, describing the returned index buffer.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if the argument is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DIndexBuffer8::Lock

Locks a range of index data and obtains a pointer to the index buffer memory.

```
HRESULT Lock(
  UINT OffsetToLock,
  UINT SizeToLock,
  BYTE** ppbData,
  DWORD Flags
);
```

Parameters

OffsetToLock
[in] Offset into the index data to lock, in bytes.

SizeToLock
[in] Size of the index data to lock, in bytes.

ppbData
[out] Address of a pointer to an array of **BYTE** values, filled with the returned index data.

Flags
[in] A combination of zero or more locking flags, describing how the index buffer memory should be locked.

D3DLOCK_DISCARD
The application overwrites, with a write-only operation, the entire index buffer. This enables Direct3D to return a pointer to a new memory area so that the dynamic memory access (DMA) and rendering from the old area do not stall.

D3DLOCK_NOOVERWRITE
Indicates that no indices that were referred to in drawing calls since the start of the frame or the last lock without this flag will be modified during the lock. This can enable optimizations when the application is only appending data to the index buffer.

D3DLOCK_NOSYSLOCK
The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long-duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY
The application will not write to the buffer. This enables some optimization. D3DLOCK_READONLY cannot be specified with D3DLOCK_DISCARD;

nor can it be specified on a vertex buffer created with D3DUSAGE_WRITEONLY.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

When working with index buffers, you are allowed to make multiple lock calls. However, you must ensure that the number of lock calls match the number of unlock calls. DrawPrimitive calls will not succeed with any outstanding lock count on any currently set index buffer.

The D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE flags are valid only on buffers created with D3DUSAGE_DYNAMIC.

See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DIndexBuffer8::Unlock

IDirect3DIndexBuffer8::Unlock

Unlocks index data.

```
HRESULT Unlock();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DIndexBuffer8::Lock

IDirect3DResource8

Applications use the methods of the **IDirect3DResource8** interface to query and prepare resources.

To create a texture resource, you can call one of the following methods.

- **IDirect3DDevice8::CreateCubeTexture**
- **IDirect3DDevice8::CreateTexture**
- **IDirect3DDevice8::CreateVolumeTexture**

To create a geometry-oriented resource, you can call one of the following methods.

- **IDirect3DDevice8::CreateIndexBuffer**
- **IDirect3DDevice8::CreateVertexBuffer**

The methods of the **IDirect3DResource8** interface can be organized into the following groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The **IDirect3DResource8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DRESOURCE8** and **PDIRECT3DRESOURCE8** types are defined as pointers to the **IDirect3DResource8** interface.

```
typedef struct IDirect3DResource8 *LPDIRECT3DRESOURCE8, *PDIRECT3DRESOURCE8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DResource8::FreePrivateData

Frees the specified private data associated with this resource.

```
HRESULT FreePrivateData(
  REFGUID refguid
);
```

Parameters

refguid
[in] Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to free.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTFOUND

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DResource8**.

- **IDirect3DBaseTexture8**
- **IDirect3DCubeTexture8**
- **IDirect3DIndexBuffer8**
- **IDirect3DTexture8**
- **IDirect3DVertexBuffer8**
- **IDirect3DVolumeTexture8**

Remarks

Microsoft® Direct3D® calls this method automatically when a resource is released.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DResource8::GetPrivateData, **IDirect3DResource8::SetPrivateData**

IDirect3DResource8::GetDevice

Retrieves the device associated with a resource.

```
HRESULT GetDevice(
  IDirect3DDevice8** ppDevice
);
```

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface to fill with the device pointer, if the query succeeds.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DResource8**.

- **IDirect3DBaseTexture8**
- **IDirect3DCubeTexture8**
- **IDirect3DIndexBuffer8**
- **IDirect3DTexture8**
- **IDirect3DVertexBuffer8**
- **IDirect3DVolumeTexture8**

Remarks

This method allows navigation to the owning device object.

Note

Calling this method will increase the internal reference count on the **IDirect3DDevice8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DDevice8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DResource8::GetPriority

Retrieves the priority for this resource.

```
DWORD GetPriority();
```

Parameters

None.

Return Values

Returns a **DWORD** value, indicating the priority of the resource.

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DResource8**.

- **IDirect3DBaseTexture8**
- **IDirect3DCubeTexture8**
- **IDirect3DIndexBuffer8**
- **IDirect3DTexture8**
- **IDirect3DVertexBuffer8**
- **IDirect3DVolumeTexture8**

Remarks

GetPriority is used for priority control of managed resources. This method returns 0 on nonmanaged resources.

Priorities are used to determine when managed resources are to be removed from memory. A resource assigned a low priority is removed before a resource with a high priority. If two resources have the same priority, the resource that was used more recently is kept in memory; the other resource is removed. Managed resources have a default priority of 0.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DResource8::SetPriority

IDirect3DResource8::GetPrivateData

Copies the private data associated with the resource to a provided buffer.

```
HRESULT GetPrivateData(
  REFGUID refguid,
  void* pData,
  DWORD* pSizeOfData
);
```

Parameters

refguid
[in] Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to retrieve.

pData
[out] Pointer to a previously allocated buffer to fill with the requested private data if the call succeeds. The application calling this method is responsible for allocating and releasing this buffer.

pSizeOfData
[in, out] Pointer to the size of the buffer at *pData*, in bytes. If this value is less than the actual size of the private data (such as 0), the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_MOREDATA

D3DERR_NOTFOUND

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DResource8**.

- **IDirect3DBaseTexture8**
- **IDirect3DCubeTexture8**
- **IDirect3DIndexBuffer8**
- **IDirect3DTexture8**
- **IDirect3DVertexBuffer8**
- **IDirect3DVolumeTexture8**

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DResource8::FreePrivateData, **IDirect3DResource8::SetPrivateData**

IDirect3DResource8::GetType

Returns the type of the resource.

```
D3DRESOURCETYPE GetType();
```

Parameters

None.

Return Values

Returns a member of the **D3DRESOURCETYPE** enumerated type, identifying the type of the resource.

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DResource8**.

- **IDirect3DBaseTexture8**

- **IDirect3DCubeTexture8**
- **IDirect3DIndexBuffer8**
- **IDirect3DTexture8**
- **IDirect3DVertexBuffer8**
- **IDirect3DVolumeTexture8**

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DResource8::PreLoad

Preloads a managed resource.

```
void PreLoad();
```

Parameters

None.

Return Values

None.

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DResource8**.

- **IDirect3DBaseTexture8**
- **IDirect3DCubeTexture8**
- **IDirect3DIndexBuffer8**
- **IDirect3DTexture8**
- **IDirect3DVertexBuffer8**
- **IDirect3DVolumeTexture8**

Remarks

Calling this method indicates that the application will need this managed resource shortly. This method has no effect on nonmanaged resources.

PreLoad detects "thrashing" conditions where more resources are being used in each frame than can fit in video memory simultaneously. Under such circumstances **Preload** silently does nothing.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DResource8::SetPriority

Assigns the resource-management priority for this resource.

```
DWORD SetPriority(
  DWORD PriorityNew
);
```

Parameters

PriorityNew
[in] **DWORD** value that specifies the new resource-management priority for the resource.

Return Values

Returns the previous priority value for the resource.

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DResource8**.

- **IDirect3DBaseTexture8**
- **IDirect3DCubeTexture8**
- **IDirect3DIndexBuffer8**
- **IDirect3DTexture8**
- **IDirect3DVertexBuffer8**
- **IDirect3DVolumeTexture8**

Remarks

SetPriority is used for priority control of managed resources. This method returns 0 on non-managed resources.

Priorities are used to determine when managed resources are to be removed from memory. A resource assigned a low priority is removed before a resource with a high

priority. If two resources have the same priority, the resource that was used more recently is kept in memory; the other resource is removed. Managed resources have a default priority of 0.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DResource8::GetPriority

IDirect3DResource8::SetPrivateData

Associates data with the resource that is intended for use by the application, not by Microsoft® Direct3D®. Data is passed by value, and multiple sets of data can be associated with a single resource.

```
HRESULT SetPrivateData(
  REFGUID refguid,
  CONST void* pData,
  DWORD SizeOfData,
  DWORD Flags
);
```

Parameters

refguid
[in] Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to set.

pData
[in] Pointer to a buffer that contains the data to be associated with the resource.

SizeOfData
[in] Size of the buffer at *pData*, in bytes.

Flags
[in] Value that describes the type of data being passed, or indicates to the application that the data should be invalidated when the resource changes.

(none)
If no flags are specified, Direct3D® allocates memory to hold the data within the buffer and copies the data into the new buffer. The buffer allocated by Direct3D is automatically freed, as appropriate.

D3DSPD_IUNKNOWN
The data at *pData* is a pointer to an **IUnknown** interface. *SizeOfData* must be set to the size of a pointer to an **IUnknown** interface, sizeof(IUnknown*). Direct3D automatically calls **IUnknown::AddRef** through *pData* and

IUnknown::Release when the private data is destroyed. Private data will be destroyed by a subsequent call to **SetPrivateData** with the same GUID, a subsequent call to **IDirect3DResource8::FreePrivateData**, or when the Direct3D8 object is released. For more information, see Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Applies To

This method applies to the following interfaces, which inherit from **IDirect3DResource8**.

- **IDirect3DBaseTexture8**
- **IDirect3DCubeTexture8**
- **IDirect3DIndexBuffer8**
- **IDirect3DTexture8**
- **IDirect3DVertexBuffer8**
- **IDirect3DVolumeTexture8**

Remarks

Direct3D does not manage the memory at *pData*. If this buffer was dynamically allocated, it is the caller's responsibility to free the memory.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DResource8::FreePrivateData, **IDirect3DResource8::GetPrivateData**

IDirect3DSurface8

Applications use the methods of the **IDirect3DSurface8** interface to query and prepare surfaces.

The methods of the **IDirect3DSurface8** interface can be organized into the following groups.

Devices	**GetDevice**
Information	**GetContainer**
	GetDesc
Locking Surfaces	**LockRect**
	UnlockRect
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData

The **IDirect3DSurface8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DSURFACE8** and **PDIRECT3DSURFACE8** types are defined as pointers to the **IDirect3DSurface8** interface.

```
typedef struct IDirect3DSurface8 *LPDIRECT3DSURFACE8, *PDIRECT3DSURFACE8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

Surface Interfaces

IDirect3DSurface8::FreePrivateData

Frees the specified private data associated with this surface.

```
HRESULT FreePrivateData(
  REFGUID refguid
);
```

Parameters

refguid

[in] Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to free.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTFOUND

Remarks

Microsoft® Direct3D® calls this method automatically when a surface is released.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DSurface8::GetPrivateData, **IDirect3DSurface8::SetPrivateData**

IDirect3DSurface8::GetContainer

Provides access to the parent cube texture or texture (mipmap) object, if this surface is a child level of a cube texture or a mipmap.

```
HRESULT GetContainer(
  REFIID riid
  void** ppContainer
);
```

Parameters

riid
[in] Reference identifier of the container being requested.

ppContainer
[out] Address of a pointer to fill with the container pointer if the query succeeds. See Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

If the surface is created using **IDirect3DDevice8::CreateImageSurface**, **IDirect3DDevice8::CreateRenderTarget**, or **IDirect3DDevice8::CreateDepthStencilSurface**, the surface is considered stand alone. In this case, **GetContainer** will return the Direct3D device used to create the surface.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DSurface8::GetDevice

Retrieves the device associated with a surface.

```
HRESULT GetDevice(
  IDirect3DDevice8** ppDevice
);
```

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface to fill with the device pointer, if the query succeeds.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method allows navigation to the owning device object.

Note

Calling this method will increase the internal reference count on the **IDirect3DDevice8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DDevice8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DSurface8::GetPrivateData

Copies the private data associated with the surface to a provided buffer.

```
HRESULT GetPrivateData(
  REFGUID refguid,
  void* pData,
  DWORD* pSizeOfData
);
```

Parameters

refguid
[in] Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to retrieve.

pData
[in, out] Pointer to a previously allocated buffer to fill with the requested private data if the call succeeds. The application calling this method is responsible for allocating and releasing this buffer.

pSizeOfData
[in, out] Pointer to the size of the buffer at *pData*, in bytes. If this value is less than the actual size of the private data (such as 0), the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_MOREDATA

D3DERR_NOTFOUND

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DSurface8::FreePrivateData, **IDirect3DSurface8::SetPrivateData**

IDirect3DSurface8::GetDesc

Retrieves a description of the surface.

```
HRESULT GetDesc(
  D3DSURFACE_DESC* pDesc
);
```

Parameters

pDesc
[out] Pointer to a **D3DSURFACE_DESC** structure, describing the surface.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if the argument is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DSurface8::LockRect

Locks a rectangle on a surface.

```
HRESULT LockRect(
  D3DLOCKED_RECT* pLockedRect,
  CONST RECT* pRect,
  DWORD Flags
);
```

Parameters

pLockedRect
[out] Pointer to a **D3DLOCKED_RECT** structure, describing the locked region.

pRect
[in] Pointer to a rectangle to lock. Specified by a pointer to a **RECT** structure. Specifying NULL for this parameter expands the dirty region to cover the entire surface.

Flags

[in] A combination of zero or more locking flags, describing the type of lock to perform.

D3DLOCK_NO_DIRTY_UPDATE
By default, a lock on a resource adds a dirty region to that resource. This flag prevents any changes to the dirty state of the resource. Applications should use this flag when they have additional information about the actual set of regions changed during the lock operation.

D3DLOCK_NOSYSLOCK
The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long-duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY
The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The only lockable format for a depth-stencil surface is D3DFMT_D16_LOCKABLE.

A multisample backbuffer cannot be locked.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DSurface8::UnlockRect

IDirect3DSurface8::SetPrivateData

Associates data with the surface that is intended for use by the application, not by Microsoft® Direct3D®.

HRESULT SetPrivateData(

```
  REFGUID refguid,
  void* pData,
  DWORD SizeOfData,
  DWORD Flags
);
```

Parameters

refguid

[in] Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to set.

pData

[in] Pointer to a buffer that contains the data to associate with the surface.

SizeOfData

[in] Size of the buffer at *pData*, in bytes.

Flags

[in] Value that describes the type of data being passed, or indicates to the application that the data should be invalidated when the resource changes.

(none)

If no flags are specified, Direct3D allocates memory to hold the data within the buffer and copies the data into the new buffer. The buffer allocated by Direct3D is automatically freed, as appropriate.

D3DSPD_IUNKNOWN

The data at *pData* is a pointer to an **IUnknown** interface. *SizeOfData* must be set to the size of a pointer to an **IUnknown** interface, sizeof(IUnknown*). Direct3D automatically calls **IUnknown::AddRef** through *pData* and **IUnknown::Release** when the private data is destroyed. Private data will be destroyed by a subsequent call to **SetPrivateData** with the same GUID, a subsequent call to **IDirect3DSurface8::FreePrivateData**, or when the Direct3D8 object is released. For more information, see Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

Direct3D does not manage the memory at *pData*. If this buffer was dynamically allocated, it is the caller's responsibility to free the memory.

Data is passed by value, and multiple sets of data can be associated with a single surface.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DSurface8::FreePrivateData, **IDirect3DSurface8::GetPrivateData**

IDirect3DSurface8::UnlockRect

Unlocks a rectangle on a surface.

```
HRESULT UnlockRect();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DSurface8::LockRect

IDirect3DSwapChain8

Applications use the methods of the **IDirect3DSwapChain8** interface to manipulate a swap chain.

There is always at least one swap chain for each device, known as the implicit swap chain. However, an additional swap chain for rendering multiple views from the same device can be created by calling the **IDirect3DDevice8::CreateAdditionalSwapChain** method.

The methods of the **IDirect3DSwapChain8** interface can be organized into the following groups.

Presentation	**Present**
Surface Management	**GetBackBuffer**

The **IDirect3DSwapChain8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DSWAPCHAIN8** and **PDIRECT3DSWAPCHAIN8** types are defined as pointers to the **IDirect3DSwapChain8** interface.

```
typedef struct IDirect3DSwapChain8 *LPDIRECT3DSWAPCHAIN8,
*PDIRECT3DSWAPCHAIN8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CreateAdditionalSwapChain

IDirect3DSwapChain8::GetBackBuffer

Retrieves a back buffer from the swap chain of the device.

```
HRESULT GetBackBuffer(
  UINT  BackBuffer,
  D3DBACKBUFFER_TYPE Type,
  IDirect3DSurface8** ppBackBuffer
);
```

Parameters

BackBuffer

[in] Index of the back buffer object to return. Back buffers are numbered from 0 to the total number of back buffers - 1. A value of 0 returns the first back buffer, not the front buffer. The front buffer is not accessible through this method.

Type

[in] Stereo view is not supported in DirectX 8.0, so the only valid value for this parameter is D3DBACKBUFFER_TYPE_MONO.

ppBackBuffer

[out, retval] Address of a pointer to an **IDirect3DSurface8** interface, representing the returned back buffer surface.

Return Values

If the method succeeds, the return value is D3D_OK.

If *BackBuffer* exceeds or equals the total number of back buffers, then the function fails and returns D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count on the **IDirect3DSurface8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DSurface8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DSwapChain8::Present

Presents the contents of the next in the sequence of back buffers owned by the swap chain.

```
HRESULT Present(
  CONST RECT* pSourceRect,
  CONST RECT* pDestRect,
  HWND hDestWindowOverride,
  CONST RGNDATA* pDirtyRegion
);
```

Parameters

pSourceRect

[in] Pointer to a **RECT** structure that contains the source rectangle. This value must be NULL unless the swap chain was created with D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC. If NULL, the entire source surface is presented. If the rectangle exceeds the source surface, the rectangle is clipped to the source surface.

pDestRect

[in] Pointer to a **RECT** structure containing the destination rectangle, in window client coordinates. This value must be NULL unless the swap chain was created

with D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC. If NULL, the entire client area is filled. If the rectangle exceeds the destination client area, the rectangle is clipped to the destination client area.

hDestWindowOverride
[in] Destination window whose client area is taken as the target for this presentation. If this value is NULL, the *hWndDeviceWindow* member of **D3DPRESENT_PARAMETERS** is taken.

pDirtyRegion
[in] This parameter is not used and must be set to NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

This method performs identically to the **IDirect3DDevice8::Present** method.

If necessary, a stretch operation is applied to transfer the pixels within the source rectangle to the destination rectangle in the client area of the target window.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::Reset

IDirect3DTexture8

Applications use the methods of the **IDirect3DTexture8** interface to manipulate a texture resource.

The **IDirect3DTexture8** interface is obtained by calling the **IDirect3DDevice8::CreateTexture** method.

The **IDirect3DTexture8** interface inherits the following **IDirect3DResource8** methods, which can be organized into these groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The **IDirect3DTexture8** interface inherits the following **IDirect3DBaseTexture8** methods, which can be organized into these groups.

Detail	**GetLOD**
	SetLOD
Information	**GetLevelCount**

The methods of the **IDirect3DTexture8** interface can be organized into the following groups.

Information	**GetLevelDesc**
Locking Surfaces	**LockRect**
	UnlockRect
Miscellaneous	**AddDirtyRect**
	GetSurfaceLevel

The **IDirect3DTexture8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DTEXTURE8** and **PDIRECT3DTEXTURE8** types are defined as pointers to the **IDirect3DTexture8** interface.

```
typedef struct IDirect3DTexture8 *LPDIRECT3DTEXTURE8, *PDIRECT3DTEXTURE8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CreateTexture

IDirect3DTexture8::AddDirtyRect

Adds a dirty region to a texture resource.

```
HRESULT AddDirtyRect(
  CONST RECT* pDirtyRect
);
```

Parameters

pDirtyRect
[in] Pointer to a **RECT** structure, specifying the dirty region to add. Specifying NULL expands the dirty region to cover the entire texture.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Applications can optimize what subset of a resource is copied by specifying dirty regions on the resource.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DTexture8::GetLevelDesc

Retrieves a level description of a texture resource.

```
HRESULT GetLevelDesc(
  UINT Level,
  D3DSURFACE_DESC* pDesc
);
```

Parameters

Level

[in] Identifies a level of the texture resource. This method returns a surface description for the level specified by this parameter.

pDesc

[out] Pointer to a **D3DSURFACE_DESC** structure, describing the returned level.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if one of the arguments is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DTexture8::GetSurfaceLevel

Retrieves the specified texture surface level.

```
HRESULT GetSurfaceLevel(
  UINT Level,
  IDirect3DSurface8** ppSurfaceLevel
);
```

Parameters

Level

[in] Identifies a level of the texture resource. This method returns a surface for the level specified by this parameter. The top-level surface is denoted by 0.

ppSurfaceLevel

[out, retval] Address of a pointer to an **IDirect3DSurface8** interface, representing the returned surface.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count on the **IDirect3DSurface8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DSurface8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DTexture8::LockRect

Locks a rectangle on a texture resource.

```
HRESULT LockRect(
  UINT Level,
  D3DLOCKED_RECT* pLockedRect,
  CONST RECT* pRect,
  DWORD Flags
);
```

Parameters

Level
[in] Specifies the level of the texture resource to lock.

pLockedRect
[out] Pointer to a **D3DLOCKED_RECT** structure, describing the locked region.

pRect
[in] Pointer to a rectangle to lock. Specified by a pointer to a **RECT** structure. Specifying NULL for this parameter expands the dirty region to cover the entire texture.

Flags
[in] A combination of zero or more locking flags, describing the type of lock to perform.

D3DLOCK_NO_DIRTY_UPDATE
By default, a lock on a resource adds a dirty region to that resource. This flag prevents any changes to the dirty state of the resource. Applications should use this flag when they have additional information about the actual set of regions changed during the lock operation and can then pass this information to **IDirect3DTexture8::AddDirtyRect**.

D3DLOCK_NOSYSLOCK
The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long-duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY

The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Textures created in D3DPOOL_DEFAULT are not-lockable.

The only lockable format for a depth-stencil surface is D3DFMT_D16_LOCKABLE.

A multisample backbuffer cannot be locked. Video memory textures cannot be locked, but must be modified by calling **IDirect3Ddevice8::CopyRects** or **IDirect3Ddevice8::UpdateTexture**. There are exceptions for some proprietary driver pixel formats that Microsoft® DirectX® 8.0 does not recognize. These can be locked.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DTexture8::UnlockRect

IDirect3DTexture8::UnlockRect

Unlocks a rectangle on a texture resource.

```
HRESULT UnlockRect(
  UINT Level
);
```

Parameters

Level
[in] Specifies the level of the texture resource to unlock.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DTexture8::LockRect

IDirect3DVertexBuffer8

Applications use the methods of the **IDirect3DVertexBuffer8** interface to manipulate vertex buffer resources.

The **IDirect3DVertexBuffer8** interface is obtained by calling the **IDirect3DDevice8::CreateVertexBuffer** method.

The **IDirect3DVertexBuffer8** interface inherits the following **IDirect3DResource8** methods, which can be organized into these groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The methods of the **IDirect3DVertexBuffer8** interface can be organized into the following groups.

Information	**GetDesc**
Locking	**Lock**
	Unlock

The **IDirect3DVertexBuffer8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DVERTEXBUFFER8** and **PDIRECT3DVERTEXBUFFER8** types are defined as pointers to the **IDirect3DVertexBuffer8** interface.

```
typedef struct IDirect3DVertexBuffer8 *LPDIRECT3DVERTEXBUFFER8,
*PDIRECT3DVERTEXBUFFER8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CreateVertexBuffer

IDirect3DVertexBuffer8::GetDesc

Retrieves a description of the vertex buffer resource.

```
HRESULT GetDesc(
  D3DVERTEXBUFFER_DESC* pDesc
);
```

Parameters

pDesc
[out] Pointer to a **D3DVERTEXBUFFER_DESC** structure, describing the returned vertex buffer.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if the argument is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DVertexBuffer8::Lock

Locks a range of vertex data and obtains a pointer to the vertex buffer memory.

```
HRESULT Lock(
  UINT OffsetToLock,
  UINT SizeToLock,
```

```
  BYTE** ppbData,
  DWORD Flags
);
```

Parameters

OffsetToLock
[in] Offset into the vertex data to lock, in bytes.

SizeToLock
[in] Size of the vertex data to lock, in bytes. Specify 0 to lock the entire vertex buffer.

ppbData
[out] Address of a pointer to an array of **BYTE** values, filled with the returned vertex data.

Flags
[in] A combination of zero or more locking flags, indicating how the vertex buffer memory should be locked.

D3DLOCK_DISCARD
The application overwrites, with a write-only operation, the entire index buffer. This enables Direct3D to return a pointer to a new memory area so that the dynamic memory access (DMA) and rendering from the old area do not stall. See remarks.

D3DLOCK_NOOVERWRITE
Indicates that no vertices that were referred to in drawing calls since the start of the frame or the last lock without this flag will be modified during the lock. This can enable optimizations when the application is appending data only to the vertex buffer. See Remarks.

D3DLOCK_NOSYSLOCK
The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long-duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY
The application will not write to the buffer. This enables some optimizations. D3DLOCK_READONLY cannot be specified with D3DLOCK_DISCARD, nor can it be specified on a vertex buffer created with D3DUSAGE_WRITEONLY.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

When working with vertex buffers, you are allowed to make multiple lock calls; however, you must ensure that the number of lock calls match the number of unlock calls. DrawPrimitive calls will not succeed with any outstanding lock count on any currently set vertex buffer.

The D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE flags are valid only on buffers created with D3DUSAGE_DYNAMIC.

See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVertexBuffer8::Unlock

IDirect3DVertexBuffer8::Unlock

Unlocks vertex data.

```
HRESULT Unlock();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVertexBuffer8::Lock

IDirect3DVolume8

Applications use the methods of the **IDirect3DVolume8** interface to manipulate volume resources.

The **IDirect3DVolume8** interface is obtained by calling the **IDirect3DVolumeTexture8::GetVolumeLevel** method.

The methods of the **IDirect3DVolume8** interface can be organized into the following groups.

Devices	**GetDevice**
Information	**GetContainer**
	GetDesc
Locking Volumes	**LockBox**
	UnlockBox
Private Volume Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData

The **IDirect3DVolume8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DVOLUME8** and **PDIRECT3DVOLUME8** types are defined as pointers to the **IDirect3DVolume8** interface.

```
typedef struct IDirect3DVolume8 *LPDIRECT3DVOLUME8, *PDIRECT3DVOLUME8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DVolume8::FreePrivateData

Frees the specified private data associated with this volume.

```
HRESULT FreePrivateData(
  REFGUID refguid
);
```

Parameters

refguid
[in] Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to free.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTFOUND

Remarks

Microsoft® Direct3D® calls this method automatically when a volume is released.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVolume8::GetPrivateData, **IDirect3DVolume8::SetPrivateData**

IDirect3DVolume8::GetContainer

Provides access to the parent volume texture object, if this surface is a child level of a volume texture.

```
HRESULT GetContainer(
  REFIID riid,
  void** ppContainer
);
```

Parameters

riid
[in] Reference identifier of the volume being requested.

ppContainer
[out, retval] Address of a pointer to fill with the container pointer, if the query succeeds.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DVolume8::GetDevice

Retrieves the device associated with a volume.

```
HRESULT GetDevice(
  IDirect3DDevice8** ppDevice
);
```

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface to fill with the device pointer, if the query succeeds.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method allows navigation to the owning device object.

Note
Calling this method will increase the internal reference count on the **IDirect3DDevice8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DDevice8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DVolume8::GetPrivateData

Copies the private data associated with the volume to a provided buffer.

```
HRESULT GetPrivateData(
  REFGUID refguid,
  void* pData,
  DWORD* pSizeOfData
);
```

Parameters

refguid
[in] Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to retrieve.

pData
[in, out] Pointer to a previously allocated buffer to fill with the requested private data if the call succeeds. The application calling this method is responsible for allocating and releasing this buffer.

pSizeOfData
[in, out] Pointer to the size of the buffer at *pData*, in bytes. If this value is less than the actual size of the private data, such as 0, the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_MOREDATA

D3DERR_NOTFOUND

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVolume8::FreePrivateData, **IDirect3DVolume8::SetPrivateData**

IDirect3DVolume8::GetDesc

Retrieves a description of the volume.

```
HRESULT GetDesc(
  D3DVOLUME_DESC* pDesc
);
```

Parameters

pDesc
[out] Pointer to a **D3DVOLUME_DESC** structure, describing the volume.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if the argument is invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DVolume8::LockBox

Locks a box on a volume resource.

```
HRESULT LockBox(
  D3DLOCKED_BOX* pLockedVolume,
  CONST D3DBOX* pBox,
  DWORD Flags
);
```

Parameters

pLockedVolume
[out] Pointer to a **D3DLOCKED_BOX** structure, describing the locked region.

pBox
[in] Pointer to a box to lock. Specified by a pointer to a **D3DBOX** structure. Specifying NULL for this parameter locks the entire volume.

Flags

[in] A combination of zero or more locking flags, describing the type of lock to perform.

D3DLOCK_NO_DIRTY_UPDATE
By default, a lock on a resource adds a dirty region to that resource. This flag prevents any changes to the dirty state of the resource. Applications should use this flag when they have additional information about the actual set of volumes changed during the lock operation.

D3DLOCK_NOSYSLOCK
The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long-duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY
The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVolume8::UnlockBox

IDirect3DVolume8::SetPrivateData

Associates data with the volume that is intended for use by the application, not by Microsoft® Direct3D®.

```
HRESULT SetPrivateData(
  REFGUID refguid,
  void* pData,
  DWORD SizeOfData,
  DWORD Flags
);
```

Parameters

refguid
[in] Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to set.

pData
[in] Pointer to a buffer that contains the data to associate with the volume.

SizeOfData
[in] Size of the buffer at *pData*, in bytes.

Flags
[in] Value that describes the type of data being passed, or indicates to the application that the data should be invalidated when the resource changes.

(none)
If no flags are specified, Direct3D allocates memory to hold the data within the buffer and copies the data into the new buffer. The buffer allocated by Direct3D is automatically freed, as appropriate.

D3DSPD_IUNKNOWN
The data at *pData* is a pointer to an **IUnknown** interface. *SizeOfData* must be set to the size of a pointer to an **IUnknown** interface, sizeof(IUnknown*). Direct3D automatically calls **IUnknown::AddRef** through *pData* and **IUnknown::Release** when the private data is destroyed. Private data will be destroyed by a subsequent call to **SetPrivateData** with the same GUID, a subsequent call to **IDirect3DVolume8::FreePrivateData**, or when the Direct3D8 object is released. For more information, see Remarks.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

Direct3D does not manage the memory at *pData*. If this buffer was dynamically allocated, it is the caller's responsibility to free the memory.

Data is passed by value, and multiple sets of data can be associated with a single volume.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVolume8::FreePrivateData, **IDirect3DVolume8::GetPrivateData**

IDirect3DVolume8::UnlockBox

Unlocks a box on a volume resource.

```
HRESULT UnlockBox();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVolume8::LockBox

IDirect3DVolumeTexture8

Applications use the methods of the **IDirect3DVolumeTexture8** interface to manipulate a volume texture resource.

The **IDirect3DVolumeTexture8** interface is obtained by calling the **IDirect3DDevice8::CreateVolumeTexture** method.

The **IDirect3DVolumeTexture8** interface inherits the following **IDirect3DResource8** methods, which can be organized into these groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**

	PreLoad
	SetPriority

The **IDirect3DVolumeTexture8** interface inherits the following **IDirect3DBaseTexture8** methods, which can be organized into these groups.

Detail	**GetLOD**
	SetLOD
Information	**GetLevelCount**

The methods of the **IDirect3DVolumeTexture8** interface can be organized into the following groups.

Information	**GetLevelDesc**
Locking Volumes	**LockBox**
	UnlockBox
Miscellaneous	**AddDirtyBox**
	GetVolumeLevel

The **IDirect3DVolumeTexture8** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DVOLUMETEXTURE8** and **PDIRECT3DVOLUMETEXTURE8** types are defined as pointers to the **IDirect3DVolumeTexture8** interface.

```
typedef struct IDirect3DVolumeTexture8 *LPDIRECT3DVOLUMETEXTURE8,
*PDIRECT3DVOLUMETEXTURE8;
```

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DDevice8::CreateVolumeTexture

IDirect3DVolumeTexture8::AddDirtyBox

Adds a dirty region to a volume texture resource.

```
HRESULT AddDirtyBox(
  CONST D3DBOX* pDirtyBox
);
```

Parameters

pDirtyBox
[in] Pointer to a **D3DBOX** structure, specifying the dirty region to add. Specifying NULL expands the dirty region to cover the entire volume texture.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Applications can optimize what subset of a resource is copied by specifying boxes on the resource. However, the dirty regions may be expanded to optimize alignment.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DVolumeTexture8::GetLevelDesc

Retrieves a level description of a volume texture resource.

```
HRESULT GetLevelDesc(
  UINT Level,
  D3DVOLUME_DESC* pDesc
);
```

Parameters

Level
[in] Identifies a level of the volume texture resource. This method returns a volume description for the level specified by this parameter.

pDesc
[out] Pointer to a **D3DVOLUME_DESC** structure, describing the returned volume texture level.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL is returned if one or more of the arguments are invalid.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DVolumeTexture8::GetVolumeLevel

Retrieves the specified volume texture level.

```
HRESULT GetVolumeLevel(
  UINT Level,
  IDirect3DVolume8** ppVolumeLevel
);
```

Parameters

Level
[in] Identifies a level of the volume texture resource. This method returns a volume for the level specified by this parameter.

ppVolumeLevel
[out, retval] Address of a pointer to an **IDirect3DVolume8** interface, representing the returned volume level.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count on the **IDirect3DVolume8** interface. Failure to call **IUnknown::Release** when finished using this **IDirect3DVolume8** interface results in a memory leak.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

IDirect3DVolumeTexture8::LockBox

Locks a box on a volume texture resource.

```
HRESULT LockBox(
  UINT Level,
  D3DLOCKED_BOX* pLockedVolume,
  CONST D3DBOX* pBox,
  DWORD Flags
);
```

Parameters

Level

[in] Specifies the level of the volume texture resource to lock.

pLockedVolume

[out] Pointer to a **D3DLOCKED_BOX** structure, describing the locked region.

pBox

[in] Pointer to the volume to lock. This parameter is specified by a pointer to a **D3DBOX** structure. Specifying NULL for this parameter locks the entire volume level.

Flags

[in] A combination of zero or more locking flags, describing the type of lock to perform.

D3DLOCK_NO_DIRTY_UPDATE

By default, a lock on a resource adds a dirty region to that resource. This flag prevents any changes to the dirty state of the resource. Applications should use this flag when they have additional information about the actual set of volumes changed during the lock operation and can then pass this information to **IDirect3DVolumeTexture8::AddDirtyBox**.

D3DLOCK_NOSYSLOCK

The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long-duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY

The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVolumeTexture8::UnlockBox

IDirect3DVolumeTexture8::UnlockBox

Unlocks a box on a volume texture resource.

```
HRESULT UnlockBox(
  UINT Level
);
```

Parameters

Level
[in] Specifies the level of the volume texture resource to unlock.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib.

See Also

IDirect3DVolumeTexture8::LockBox

Functions

This section contains reference information for the functions that you need to use when you work with Microsoft® Direct3D®. The following function is implemented.

- **Direct3DCreate8**

Direct3DCreate8

Creates an instance of a Direct3D8 object.

```
IDirect3D8* Direct3DCreate8(
  UINT SDKVersion
);
```

Parameters

SDKVersion
The value of this parameter should be D3D_SDK_VERSION. See Remarks.

Return Values

If successful, this function returns a pointer to an **IDirect3D8** interface; otherwise, a NULL pointer is returned.

Remarks

This function creates a Direct3D8 object that supports enumeration and allows the creation of Direct3DDevice8 objects.

The D3D_SDK_VERSION identifier is passed to **Direct3DCreate8** in order to ensure that an application was built against the correct header files. This value is incremented whenever a header or other change would require applications to be rebuilt. If the version does not match, **Direct3DCreate8** will fail.

Note that calling this function samples the current set of active display adapters. If the user dynamically adds adapters, either by adding devices to the desktop or by hot-docking a laptop, then those devices will not be enumerated for the lifetime of this Direct3D8 object. Creating a new Direct3D8 object will expose the new devices.

Requirements

Header: Declared in D3d8.h.
Import Library: Use D3d8.lib

Macros

This section contains reference information for the macros provided by Microsoft® Direct3D®.

- **D3DCLIPPLANE***n*
- **D3DCOLOR_ARGB**
- **D3DCOLOR_COLORVALUE**
- **D3DCOLOR_RGBA**
- **D3DCOLOR_XRGB**
- **D3DFVF_TEXCOORDSIZE***n*
- **D3DTS_WORLD**
- **D3DTS_WORLD***n*
- **D3DTS_WORLDMATRIX**

D3DCLIPPLANE*n*

Defines bit patterns that enable user-defined clipping planes. These macros are defined as a convenience when setting values for the D3DRS_CLIPPLANEENABLE render state.

```
D3DCLIPPLANE0 (1 << 0)
D3DCLIPPLANE1 (1 << 1)
D3DCLIPPLANE2 (1 << 2)
D3DCLIPPLANE3 (1 << 3)
D3DCLIPPLANE4 (1 << 4)
D3DCLIPPLANE5 (1 << 5)
```

Parameters

None.

Remarks

User-defined clipping planes are enabled when the **DWORD** value set in the D3DRS_CLIPPLANEENABLE render state contains one or more set bits (that is, is not 0). The value of the render-state **DWORD** is not important; the system does not interpret the value as a number. Rather, it parses the **DWORD**, enabling any clipping plane whose corresponding bit is set. Bit 0 controls the state of the first clipping plane (at index 0), bit 1 the second plane, and so on.

The bit patterns that these macros create can be combined by using a logical **OR** operation to simultaneously enable multiple clipping planes. Omitting one of these macros from the combination effectively disables the clipping plane at that index.

Requirements

Header: Declared in D3d8types.h.

D3DCOLOR_ARGB

Initializes a color with the supplied alpha, red, green, and blue values.

```
D3DCOLOR_ARGB(a,r,g,b) \
    ((D3DCOLOR)((((a)&0xff)<<24)|(((r)&0xff)<<16)|(((g)&0xff)<<8)|((b)&0xff)))
```

Parameters

a, *r*, *g*, and *b*
Alpha, red, green, and blue components of the color. These values must be in the range 0 to 255.

Return Values

Returns the **D3DCOLOR** value that corresponds to the supplied ARGB values.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DCOLOR_RGBA, **D3DCOLOR_XRGB**

D3DCOLOR_COLORVALUE

Initializes a color with the supplied red, green, blue, and alpha floating-point values.

```
D3DCOLOR_COLORVALUE(r,g,b,a) \
    D3DCOLOR_RGBA((DWORD)((r)*255.f),(DWORD)((g)*255.f),(DWORD)
((b)*255.f),(DWORD)((a)*255.f))
```

Parameters

r, *g*, *b*, and *a*
Red, green, blue, and alpha components of the color. These values must be floating-point values in the range 0.0 through 1.0.

Return Values

Returns the **D3DCOLOR** value that corresponds to the supplied RGBA values.

Requirements

Header: Declared in D3d8types.h.

D3DCOLOR_RGBA

Initializes a color with the supplied red, green, blue, and alpha values.

D3DCOLOR_RGBA(r,g,b,a) D3DCOLOR_ARGB(a,r,g,b)

Parameters

r, *g*, *b*, and *a*
Red, green, blue, and alpha components of the color. These values must be in the range 0 through 255.

Return Values

Returns the **D3DCOLOR** value that corresponds to the supplied RGBA values.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DCOLOR_ARGB, **D3DCOLOR_XRGB**

D3DCOLOR_XRGB

Initializes a color with the supplied red, green, and blue values.

D3DCOLOR_XRGB(r,g,b) D3DCOLOR_ARGB(0xff,r,g,b)

Parameters

r, *g*, and *b*
Red, green, and blue components of the color. These values must be in the range 0 through 255.

Return Values

Returns the **D3DCOLOR** value that corresponds to the supplied RGB values.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DCOLOR_ARGB, **D3DCOLOR_RGBA**

D3DFVF_TEXCOORDSIZE*n*

Constructs bit patterns that are used to identify texture coordinate formats within a flexible vertex format description. The results of these macros can be combined within a flexible vertex format description by using the **OR** operator.

```
D3DFVF_TEXCOORDSIZE1(CoordIndex) (D3DFVF_TEXTUREFORMAT1 <<
(CoordIndex*2 + 16))
D3DFVF_TEXCOORDSIZE2(CoordIndex) (D3DFVF_TEXTUREFORMAT2)
D3DFVF_TEXCOORDSIZE3(CoordIndex) (D3DFVF_TEXTUREFORMAT3 <<
(CoordIndex*2 + 16))
D3DFVF_TEXCOORDSIZE4(CoordIndex) (D3DFVF_TEXTUREFORMAT4 <<
(CoordIndex*2 + 16))
```

Parameters

CoordIndex
Value that identifies the texture coordinate set at which the texture coordinate size (1-, 2-, 3-, or 4-dimensional) applies.

Remarks

The **D3DFVF_TEXCOORDSIZE*n*** macros use the following constants.

```
#define D3DFVF_TEXTUREFORMAT1 3     // one floating point value
#define D3DFVF_TEXTUREFORMAT2 0     // two floating point values
#define D3DFVF_TEXTUREFORMAT3 1     // three floating point values
#define D3DFVF_TEXTUREFORMAT4 2     // four floating point values
```

The following flexible vertex format description identifies a vertex format that has a position; a normal; diffuse and specular colors; and two sets of texture coordinates. The first set of texture coordinates includes a single element, and the second set includes two elements:

```
DWORD dwFVF;
dwFVF = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE | D3DFVF_SPECULAR |
    D3DFVF_TEXCOORDSIZE1(0) |  \\ Uses 1-D texture coordinates for
                \\ texture coordinate set 1 (index
                \\ 0).
```

```
D3DFVF_TEXCOORDSIZE2(1);   \\ And 2-D texture coordinates for
                           \\ texture coordinate set 2 (index
                           \\ 1).
```

Requirements

Header: Declared in D3d8types.h.

See Also

Flexible Vertex Format Flags

D3DTS_WORLD

Identifies the transformation matrix being set as the world transformation matrix.

```
D3DTS_WORLD  D3DTS_WORLDMATRIX(0)
```

Parameters

None.

Remarks

This macro is provided to facilitate porting existing applications to Microsoft® DirectX® 8.0.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::SetTransform, **D3DTS_WORLDMATRIX**

D3DTS_WORLD*n*

Identifies subsequent transformation matrices that can be used to blend vertices by using the corresponding matrix and a blending (beta) weight value specified in the vertex format.

```
D3DTS_WORLD1 D3DTS_WORLDMATRIX(1)
D3DTS_WORLD2 D3DTS_WORLDMATRIX(2)
D3DTS_WORLD3 D3DTS_WORLDMATRIX(3)
```

Parameters

None.

Remarks

These macros are provided to facilitate porting existing applications to Microsoft® DirectX® 8.0.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::SetTransform, **D3DTS_WORLDMATRIX**

D3DTS_WORLDMATRIX

Maps indices in the range 0 through 255 to the corresponding transform states.

```
D3DTS_WORLDMATRIX(index) (D3DTRANSFORMSTATETYPE)(index + 256)
```

Parameters

index
An index value in the range 0 through 255.

Remarks

Transform states in the range 256 through 511 are reserved to store up to 256 matrices that can be indexed using 8-bit indices.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::SetTransform

Vertex Shader Declarator Macros

The following section covers the bit assembly macros used to compose the vertex shader declaration **DWORD** array.

- **D3DVSD_CONST**
- **D3DVSD_END**
- **D3DVSD_NOP**
- **D3DVSD_REG**
- **D3DVSD_SKIP**
- **D3DVSD_STREAM**
- **D3DVSD_STREAM_TESS**
- **D3DVSD_TESSNORMAL**
- **D3DVSD_TESSUV**

D3DVSD_CONST

Loads data into the vertex shader constant memory.

```
D3DVSD_CONST( _ConstantAddress, _Count ) \
        (D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_CONSTMEM) | \
        ((_Count) << D3DVSD_CONSTCOUNTSHIFT) | (_ConstantAddress))
```

Parameters

_ConstantAddress
Address of the constant array to begin filling data. Possible values range from 0 to 95.

_Count
Number of constant vectors to load (4 **DWORD**s each) followed by 4**_Count* **DWORD**s of data. See Remarks.

Remarks

The following code example shows how this macro might be used.

```
D3DVALUE diffuse[] = {1, 1, 1.0f/255.0f, 1};
    DWORD decl[] =
    {
        D3DVSD_CONST(8, 1),
        *(DWORD*)&diffuse[0],
        *(DWORD*)&diffuse[1],
        *(DWORD*)&diffuse[2],
        *(DWORD*)&diffuse[3],
        D3DVSD_STREAM(0),
        D3DVSD_REG( 0, D3DVSDT_FLOAT3),
        D3DVSD_REG( 2, D3DVSDT_D3DCOLOR),
        D3DVSD_STREAM(1),
        D3DVSD_REG( 1, D3DVSDT_FLOAT1),
```

```
    D3DVSD_STREAM(2),
    D3DVSD_REG( 3, D3DVSDT_FLOAT2),
    D3DVSD_END()
};
```

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateVertexShader

D3DVSD_END

Generates an END token.

```
#define D3DVSD_END()
```

Parameters

None.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateVertexShader

D3DVSD_NOP

Generates an NOP token.

```
#define D3DVSD_NOP()
```

Parameters

None.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateVertexShader

D3DVSD_REG

Binds a single vertex register to a vertex element from the vertex stream.

```
D3DVSD_REG( _VertexRegister, _Type ) \
        (D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_STREAMDATA) | \
        ((_Type) << D3DVSD_DATATYPESHIFT) | (_VertexRegister))
```

Parameters

_VertexRegister
Address of the vertex register. Possible values range from 0 through 15. For the fixed function pipeline, the input registers have the following fixed mapping.

D3DVSDE_POSITION
Register 0.

D3DVSDE_BLENDWEIGHT
Register 1.

D3DVSDE_NORMAL
Register 2.

D3DVSDE_PSIZE
Register 3.

D3DVSDE_DIFFUSE
Register 4.

D3DVSDE_SPECULAR
Register 5.

D3DVSDE_TEXCOORD0
Register 6.

D3DVSDE_TEXCOORD1
Register 7.

D3DVSDE_TEXCOORD2
Register 8.

D3DVSDE_TEXCOORD3
Register 9.

D3DVSDE_TEXCOORD4
Register 10.

D3DVSDE_TEXCOORD5
Register 11.

D3DVSDE_TEXCOORD6
Register 12.

D3DVSDE_TEXCOORD7
Register 13.

_Type
Specifies the dimensionality and arithmetic data type. The following values are defined.

D3DVSDT_D3DCOLOR
4-D packed unsigned bytes mapped to 0.0 to 1.0 range. In double word format this is ARGB, or in byte ordering it would be B, G, R, A.

D3DVSDT_FLOAT1
1-D float expanded to (*value*, 0.0, 0.0, 1.0)

D3DVSDT_FLOAT2
2-D float expanded to (*value*, *value*, 0.0, 1.0)

D3DVSDT_FLOAT3
3-D float expanded to (*value*, *value*, *value*, 1.0)

D3DVSDT_FLOAT4
4-D float.

D3DVSDT_UBYTE4
4-D unsigned byte. In double word format this is ABGR, or in byte ordering it would be R, G, B, A.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateVertexShader

D3DVSD_SKIP

Specifies how many **DWORD**s to skip in the vertex.

```
D3DVSD_SKIP( _DWORDCount ) \
        (D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_STREAMDATA) |
0x10000000 | \
        ((_DWORDCount) << D3DVSD_SKIPCOUNTSHIFT))
```

Parameters

_DWORDCount
Number of DWORDs to skip in the vertex.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateVertexShader

D3DVSD_STREAM

Sets the current stream.

```
D3DVSD_STREAM( _StreamNumber ) \
        (D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_STREAM) |
(_StreamNumber))
```

Parameters

_StreamNumber
Specifies the stream from which to get data. Possible values range from 0 through dwMaxStreams - 1.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateVertexShader

D3DVSD_STREAM_TESS

Sets the tessellator stream.

```
D3DVSD_STREAM_TESS( ) \
    (D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_STREAM) |
(D3DVSD_STREAMTESSMASK))
```

Parameters

None.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateVertexShader

D3DVSD_TESSNORMAL

Enables tessellator-generated normals.

```
D3DVSD_TESSNORMAL( _VertexRegisterIn, _VertexRegisterOut ) \

(D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_TESSELLATOR) | \
        ((_VertexRegisterIn) << D3DVSD_VERTEXREGINSHIFT) | \
        ((0x02) << D3DVSD_DATATYPESHIFT) | (_VertexRegisterOut))
```

Parameters

_VertexRegisterIn
Address of the vertex register whose input stream will be used in normal computation. Possible values range from 0 to 15.

_VertexRegisterOut
Address of the vertex register to output the normal to. Possible values range from 0 to 15.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateVertexShader

D3DVSD_TESSUV

Enables tessellator-generated surface parameters.

```
D3DVSD_TESSUV( _VertexRegister ) \
        (D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_TESSELLATOR) |
0x10000000 | \
        ((0x01) << D3DVSD_DATATYPESHIFT) | (_VertexRegister))
```

Parameters

_VertexRegister
Address of the vertex register to output parameters. Possible values range from 0 to 15.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateVertexShader

Structures

This section contains information about the structures used with Microsoft® Direct3D®.

- **D3DADAPTER_IDENTIFIER8**
- **D3DBOX**
- **D3DCAPS8**
- **D3DCLIPSTATUS8**
- **D3DCOLORVALUE**
- **D3DDEVICE_CREATION_PARAMETERS**
- **D3DDISPLAYMODE**
- **D3DGAMMARAMP**
- **D3DINDEXBUFFER_DESC**
- **D3DLIGHT8**
- **D3DLINEPATTERN**
- **D3DLOCKED_BOX**
- **D3DLOCKED_RECT**
- **D3DMATERIAL8**
- **D3DMATRIX**
- **D3DPRESENT_PARAMETERS**
- **D3DRANGE**
- **D3DRASTER_STATUS**
- **D3DRECT**
- **D3DRECTPATCH_INFO**
- **D3DSURFACE_DESC**
- **D3DTRIPATCH_INFO**
- **D3DVECTOR**
- **D3DVERTEXBUFFER_DESC**

- **D3DVIEWPORT8**
- **D3DVOLUME_DESC**

D3DADAPTER_IDENTIFIER8

Contains information identifying the adapter.

```
typedef struct _D3DADAPTER_IDENTIFIER8 {
    char        Driver[MAX_DEVICE_IDENTIFIER_STRING];
    char        Description[MAX_DEVICE_IDENTIFIER_STRING];

#ifdef _WIN32
    LARGE_INTEGER   DriverVersion;          /* Defined for 32-bit components
*/
#else
    DWORD       DriverVersionLowPart;    /* Defined for 16-bit driver
components */
    DWORD       DriverVersionHighPart;
#endif

    DWORD       VendorId;
    DWORD       DeviceId;
    DWORD       SubSysId;
    DWORD       Revision;

    GUID        DeviceIdentifier;

    DWORD       WHQLLevel;
} D3DADAPTER_IDENTIFIER8;
```

Members

Driver and **Description**

Used for presentation to the user. They should not be used to identify particular drivers, since many different strings may be associated with the same device and driver from different vendors.

DriverVersion, **DriverVersionLowPart**, and **DriverVersionHighPart**

Identify the version of the Microsoft® Direct3D® driver. It is legal to do < and > comparisons on the 64-bit signed integer value. However, exercise caution if you use this element to identify problematic drivers. Instead, you should use **DeviceIdentifier**.

The following pseudocode example illustrates the version format encoded in this member.

```
Product = HIWORD(DriverVersion.HighPart)
Version = LOWORD(DriverVersion.HighPart)
```

```
SubVersion = HIWORD(DriverVersion.LowPart)
Build = LOWORD(DriverVersion.LowPart)
```

See the Microsoft Platform Software Development Kit (SDK) for more information on the **HIWORD** macro, the **LOWORD** macro, and the **LARGE_INTEGER** structure.

VendorId
Can be used to help identify a particular chip set. Query this member to identify the manufacturer. The value may be zero if unknown.

DeviceId
Can be used to help identify a particular chip set. Query this member to identify the type of chip set. The value may be zero if unknown.

SubSysId
Can be used to help identify a particular chip set. Query this member to identify the subsystem, typically the particular board. The value may be zero if unknown.

Revision
Can be used to help identify a particular chip set. Query this member to identify the revision level of the chip set. The value may be zero if unknown.

DeviceIdentifier
Can be queried to check changes in the driver and chip set. This GUID is a unique identifier for the driver and chip set pair. Query this member to track changes to the driver and chip set in order to generate a new profile for the graphics subsystem. **DeviceIdentifier** can also be used to identify particular problematic drivers.

WHQLLevel
Used to determine the Windows Hardware Quality Lab (WHQL) certification level for this driver and device pair. The **DWORD** is a packed date structure defining the date of the release of the most recent WHQL test passed by the driver. It is legal to perform < and > operations on this value. The following illustrates the date format.

```
Bits
31-16:   The year, a decimal number from 1999 upwards.
15-8:    The month, a decimal number from 1 to 12.
7-0:     The day, a decimal number from 1 to 31.
```

The following values are also used.

0
Not certified.

1
WHQL certified, but no date information is available.

Remarks

MAX_DEVICE_IDENTIFIER_STRING is a constant with the following definition.

```
#define MAX_DEVICE_IDENTIFIER_STRING        512
```

The **VendorId**, **DeviceId**, **SubSysId**, and **Revision** members can be used in tandem to identify particular chip sets. However, use these members with caution.

Requirements

Header: Declared in D3d8types.h.

D3DBOX

Defines a volume.

```
typedef struct _D3DBOX {
    UINT        Left;
    UINT        Top;
    UINT        Right;
    UINT        Bottom;
    UINT        Front;
    UINT        Back;
} D3DBOX;
```

Members

Left, **Top**, **Right**, **Bottom**, **Front**, and **Back**
Dimensions of the box.

Remarks

D3DBOX includes the left, top, and front edges; however, the right, bottom, and back edges are not included. For example, a box that is 100 units wide and begins at 0 (thus, including the points up to and including 99) would be expressed with a value of 0 for the **Left** member and a value of 100 for the **Right** member. Note that a value of 99 is not used for the **Right** member.

The restrictions on side ordering observed for **D3DBOX** are left to right, top to bottom, and front to back.

Requirements

Header: Declared in D3d8types.h.

D3DCAPS8

Represents the capabilities of the hardware exposed through the Microsoft® Direct3D® object.

```
typedef struct _D3DCAPS8 {
    D3DDEVTYPE        DeviceType;
    UINT              AdapterOrdinal;

    DWORD             Caps;
    DWORD             Caps2;
    DWORD             Caps3;
    DWORD             PresentationIntervals;

    DWORD             CursorCaps;

    DWORD             DevCaps;

    DWORD             PrimitiveMiscCaps;
    DWORD             RasterCaps;
    DWORD             ZCmpCaps;
    DWORD             SrcBlendCaps;
    DWORD             DestBlendCaps;
    DWORD             AlphaCmpCaps;
    DWORD             ShadeCaps;
    DWORD             TextureCaps;
    DWORD             TextureFilterCaps;
    DWORD             CubeTextureFilterCaps;
    DWORD             VolumeTextureFilterCaps;
    DWORD             TextureAddressCaps;
    DWORD             VolumeTextureAddressCaps;

    DWORD             LineCaps;

    DWORD             MaxTextureWidth, MaxTextureHeight;
    DWORD             MaxVolumeExtent;

    DWORD             MaxTextureRepeat;
    DWORD             MaxTextureAspectRatio;
    DWORD             MaxAnisotropy;
    float           MaxVertexW;

    float           GuardBandLeft;
    float           GuardBandTop;
    float           GuardBandRight;
    float           GuardBandBottom;

    float           ExtentsAdjust;
    DWORD             StencilCaps;

    DWORD             FVFCaps;
```

```
    DWORD           TextureOpCaps;
    DWORD           MaxTextureBlendStages;
    DWORD           MaxSimultaneousTextures;

    DWORD           VertexProcessingCaps;
    DWORD           MaxActiveLights;
    DWORD           MaxUserClipPlanes;
    DWORD           MaxVertexBlendMatrices;
    DWORD           MaxVertexBlendMatrixIndex;

    float           MaxPointSize;

    DWORD           MaxPrimitiveCount;
    DWORD           MaxVertexIndex;
    DWORD           MaxStreams;
    DWORD           MaxStreamStride;

    DWORD           VertexShaderVersion;
    DWORD           MaxVertexShaderConst;

    DWORD           PixelShaderVersion;
    float           MaxPixelShaderValue;
} D3DCAPS8;
```

Members

DeviceType

Member of the **D3DDEVTYPE** enumerated type. The device type for which this Direct3DDevice object was created.

AdapterOrdinal

Adapter on which this Direct3DDevice object was created. This ordinal is valid only to pass to methods of the **IDirect3D8** interface that created this Direct3DDevice object. The **IDirect3D8** interface can always be retrieved by calling **IDirect3DDevice8::GetDirect3D**.

Caps

The following driver-specific capability.

D3DCAPS_READ_SCANLINE
: Display hardware is capable of returning the current scan line.

Caps2

The following driver-specific capabilities.

D3DCAPS2_CANCALIBRATEGAMMA
: The system has a calibrator installed that can automatically adjust the gamma ramp so that the result is identical on all systems that have a calibrator. To invoke the calibrator when setting new gamma levels, use the D3DSGR_CALIBRATE flag when calling the

IDirect3DDevice8::SetGammaRamp method. Calibrating gamma ramps incurs some processing overhead and should not be used frequently.

D3DCAPS2_CANRENDERWINDOWED
The driver is capable of rendering in windowed mode.

D3DCAPS2_FULLSCREENGAMMA
The driver supports dynamic gamma ramp adjustment in full-screen mode.

D3DCAPS2_NO2DDURING3DSCENE
When the D3DCAPS2_NO2DDURING3DSCENE capability is set by the driver, it means that 2-D operations cannot be performed between calls to **IDirect3DDevice8::BeginScene** and **IDirect3DDevice8::EndScene**.

Typically, this capability is set by hardware that partitions the scene and then renders each partition in sequence. The partitioning is performed in the driver, and the hardware contains a small color and depth buffer that corresponds to the size of the image partition. Typically, on this type of rendering hardware, once each part of the image is rendered, the data in the color buffers are written to video memory and the contents of the depth-buffer are discarded. Also, note that 3-D rendering does not start until **EndScene** is encountered. Next, the scene is processed in regions. Therefore, the processing order cannot be guaranteed. For example, the first region that is processed, typically the upper-left corner of the window, might include the last triangle in the frame. This differs from more traditional graphics systems in which each command is processed sequentially in the order that it was sent. The 2-D operations are implied to occur at some fixed point in the processing. In the systems that set D3DCAPS2_NO2DDURING3DSCENE, the processing order is not guaranteed. Therefore, the display adapter might discard 2-D operations that are encountered during 3-D rendering.

In general, it is recommended that 2-D operations be performed outside of a **BeginScene** and **EndScene** pair. If 2-D operations are to be performed between a **BeginScene** and **EndScene** pair, then it is necessary to check the D3DCAPS2_NO2DDURING3DSCENE capability. If it is set, the application must expect that any 2-D operation that occurs between **BeginScene** and **EndScene** will be discarded. For more information on writing applications for systems that set D3DCAPS2_NO2DDURING3DSCENE, see Remarks.

D3DCAPS2_RESERVED
Reserved; not used.

Caps3
The following driver-specific capability.

D3DCAPS3_RESERVED
Reserved; not used.

PresentationIntervals
Bit mask of values representing what presentation swap intervals are available. For windowed mode, this value must be 0; otherwise, this value must be one of the values enumerated in the **PresentationIntervals** member of **D3DCAPS8**.

D3DPRESENT_INTERVAL_IMMEDIATE
The driver supports an immediate presentation swap interval.

D3DPRESENT_INTERVAL_ONE
The driver supports a presentation swap interval of every screen refresh.

D3DPRESENT_INTERVAL_TWO
The driver supports a presentation swap interval of every second screen refresh.

D3DPRESENT_INTERVAL_THREE
The driver supports a presentation swap interval of every third screen refresh.

D3DPRESENT_INTERVAL_FOUR
The driver supports a presentation swap interval of every fourth screen refresh.

CursorCaps
Bit mask indicating what hardware support is available for cursors.

D3DCURSORCAPS_COLOR
A full-color cursor is supported in hardware. Specifically, this flag indicates that the driver supports at least a hardware color cursor in high-resolution modes (with scan lines greater than or equal to 400).

D3DCURSORCAPS_LOWRES
A full-color cursor is supported in hardware. Specifically, this flag indicates that the driver supports a hardware color cursor in both high-resolution and low-resolution modes (with scan lines less than 400).

Direct3D for Microsoft DirectX® 8.0 does not define alpha-blending cursor capabilities.

DevCaps
Flags identifying the capabilities of the device.

D3DDEVCAPS_CANBLTSYSTONONLOCAL
Device supports blits from system-memory textures to nonlocal video-memory textures.

D3DDEVCAPS_CANRENDERAFTERFLIP
Device can queue rendering commands after a page flip. Applications do not change their behavior if this flag is set; this capability simply means that the device is relatively fast.

D3DDEVCAPS_DRAWPRIMTLVERTEX
Device exports a DrawPrimitive-aware hardware abstraction layer (HAL).

D3DDEVCAPS_EXECUTESYSTEMMEMORY
Device can use execute buffers from system memory.

D3DDEVCAPS_EXECUTEVIDEOMEMORY
Device can use execute buffers from video memory.

D3DDEVCAPS_HWRASTERIZATION
Device has hardware acceleration for scene rasterization.

D3DDEVCAPS_HWTRANSFORMANDLIGHT
Device can support transformation and lighting in hardware.

D3DDEVCAPS_NPATCHES
Device supports N patches.

D3DDEVCAPS_PUREDEVICE
Device can support rasterization, transform, lighting, and shading in hardware.

D3DDEVCAPS_QUINTICRTPATCHES
Device supports quintic béziers and B-splines.

D3DDEVCAPS_RTPATCHES
Device supports rectangular and triangular patches.

D3DDEVCAPS_RTPATCHHANDLEZERO
When this device capability is set, the hardware architecture does not require caching of any information, and uncached patches (handle zero) will be drawn as efficiently as cached ones. Note that setting D3DDEVCAPS_RTPATCHHANDLEZERO does not mean that a patch with handle zero can be drawn. A handle-zero patch can always be drawn whether this cap is set or not.

D3DDEVCAPS_SEPARATETEXTUREMEMORIES
Device is texturing from separate memory pools.

D3DDEVCAPS_TEXTURENONLOCALVIDMEM
Device can retrieve textures from non-local video memory.

D3DDEVCAPS_TEXTURESYSTEMMEMORY
Device can retrieve textures from system memory.

D3DDEVCAPS_TEXTUREVIDEOMEMORY
Device can retrieve textures from device memory.

D3DDEVCAPS_TLVERTEXSYSTEMMEMORY
Device can use buffers from system memory for transformed and lit vertices.

D3DDEVCAPS_TLVERTEXVIDEOMEMORY
Device can use buffers from video memory for transformed and lit vertices.

PrimitiveMiscCaps
General capabilities for this primitive. This member can be one or more of the following flags.

D3DPMISCCAPS_BLENDOP
Device supports the alpha-blending operations defined in the **D3DBLENDOP** enumerated type.

D3DPMISCCAPS_CLIPPLANESCALEDPOINTS
Device correctly clips scaled points of size greater than 1.0 to user-defined clipping planes.

D3DPMISCCAPS_CLIPTLVERTS
Device clips post-transformed vertex primitives.

D3DPMISCCAPS_COLORWRITEENABLE
Device supports per-channel writes for the render target color buffer through the D3DRS_COLORWRITEENABLE state.

D3DPMISCCAPS_CULLCCW
The driver supports counterclockwise culling through the D3DRS_CULLMODE state. (This applies only to triangle primitives.) This flag corresponds to the D3DCULL_CCW member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_CULLCW
The driver supports clockwise triangle culling through the D3DRS_CULLMODE state. (This applies only to triangle primitives.) This flag corresponds to the D3DCULL_CW member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_CULLNONE
The driver does not perform triangle culling. This corresponds to the D3DCULL_NONE member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_LINEPATTERNREP
The driver can handle values other than 1 in the **wRepeatFactor** member of the **D3DLINEPATTERN** structure. (This applies only to line-drawing primitives.)

D3DPMISCCAPS_MASKZ
Device can enable and disable modification of the depth buffer on pixel operations.

D3DPMISCCAPS_TSSARGTEMP
Device supports D3DTA_TEMP for temporary register.

RasterCaps
Information on raster-drawing capabilities. This member can be one or more of the following flags.

D3DPRASTERCAPS_ANISOTROPY
Device supports anisotropic filtering.

D3DPRASTERCAPS_ANTIALIASEDGES
Device can anti-alias lines forming the convex outline of objects. For more information, see D3DRS_EDGEANTIALIAS.

D3DPRASTERCAPS_COLORPERSPECTIVE
Device iterates colors perspective correct.

D3DPRASTERCAPS_DITHER
Device can dither to improve color resolution.

D3DPRASTERCAPS_FOGRANGE
Device supports range-based fog. In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene.

D3DPRASTERCAPS_FOGTABLE
Device calculates the fog value by referring to a lookup table containing fog values that are indexed to the depth of a given pixel.

D3DPRASTERCAPS_FOGVERTEX
Device calculates the fog value during the lighting operation, and interpolates the fog value during rasterization.

D3DPRASTERCAPS_MIPMAPLODBIAS
Device supports level-of-detail (LOD) bias adjustments. These bias adjustments enable an application to make a mipmap appear crisper or less sharp than it normally would. For more information about LOD bias in mipmaps, see D3DTSS_MIPMAPLODBIAS.

D3DPRASTERCAPS_PAT
The driver can perform patterned drawing lines or fills with D3DRS_LINEPATTERN for the primitive being queried.

D3DPRASTERCAPS_STRETCHBLTMULTISAMPLE
Device provides limited multisample support through a stretch-blt implementation. When this capability is set, D3DRS_MULTISAMPLEANTIALIAS cannot be turned on and off in the middle of a scene. Multisample masking cannot be performed if this flag is set.

D3DPRASTERCAPS_WBUFFER
Device supports depth buffering using w.

D3DPRASTERCAPS_WFOG
Device supports w-based fog. W-based fog is used when a perspective projection matrix is specified, but affine projections still use z-based fog. The system considers a projection matrix that contains a nonzero value in the [3] [4] element to be a perspective projection matrix.

D3DPRASTERCAPS_ZBIAS
Device supports z-bias values. These are integer values assigned to polygons that allow physically coplanar polygons to appear separate. For more information, see D3DRS_ZBIAS.

D3DPRASTERCAPS_ZBUFFERLESSHSR
Device can perform hidden-surface removal (HSR) without requiring the application to sort polygons and without requiring the allocation of a depth-buffer. This leaves more video memory for textures. The method used to perform HSR is hardware-dependent and is transparent to the application.

Z-bufferless HSR is performed if no depth-buffer surface is associated with the rendering-target surface and the depth-buffer comparison test is enabled (that is, when the state value associated with the D3DRS_ZENABLE enumeration constant is set to TRUE).

D3DPRASTERCAPS_ZFOG
Device supports z-based fog.

D3DPRASTERCAPS_ZTEST
Device can perform z-test operations. This effectively renders a primitive and indicates whether any z pixels have been rendered.

ZCmpCaps
Z-buffer comparison capabilities. This member can be one or more of the following flags.

D3DPCMPCAPS_ALWAYS
Always pass the z test.

D3DPCMPCAPS_EQUAL
Pass the z test if the new z equals the current z.

D3DPCMPCAPS_GREATER
Pass the z test if the new z is greater than the current z.

D3DPCMPCAPS_GREATEREQUAL

Pass the z test if the new z is greater than or equal to the current z.

D3DPCMPCAPS_LESS
Pass the z test if the new z is less than the current z.

D3DPCMPCAPS_LESSEQUAL
Pass the z test if the new z is less than or equal to the current z.

D3DPCMPCAPS_NEVER
Always fail the z test.

D3DPCMPCAPS_NOTEQUAL
Pass the z test if the new z does not equal the current z.

SrcBlendCaps

Source-blending capabilities. This member can be one or more of the following flags. (The RGBA values of the source and destination are indicated by the subscripts *s* and *d*.)

D3DPBLENDCAPS_BOTHINVSRCALPHA
Source blend factor is $(1–A_s, 1–A_s, 1–A_s, 1–A_s)$, and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden.

D3DPBLENDCAPS_BOTHSRCALPHA
The driver supports the D3DBLEND_BOTHSRCALPHA blend mode. (This blend mode is obsolete. For more information, see **D3DBLEND**.)

D3DPBLENDCAPS_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DPBLENDCAPS_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DPBLENDCAPS_INVDESTALPHA
Blend factor is $(1–A_d, 1–A_d, 1–A_d, 1–A_d)$.

D3DPBLENDCAPS_INVDESTCOLOR
Blend factor is $(1–R_d, 1–G_d, 1–B_d, 1–A_d)$.

D3DPBLENDCAPS_INVSRCALPHA
Blend factor is $(1–A_s, 1–A_s, 1–A_s, 1–A_s)$.

D3DPBLENDCAPS_INVSRCCOLOR
Blend factor is $(1–R_d, 1–G_d, 1–B_d, 1–A_d)$.

D3DPBLENDCAPS_ONE
Blend factor is (1, 1, 1, 1).

D3DPBLENDCAPS_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DPBLENDCAPS_SRCALPHASAT
Blend factor is (f, f, f, 1); $f = \min(A_s, 1\text{-}A_d)$.

D3DPBLENDCAPS_SRCCOLOR
Blend factor is (R_s, G_s, B_s, A_s).

D3DPBLENDCAPS_ZERO
Blend factor is (0, 0, 0, 0).

DestBlendCaps

Destination-blending capabilities. This member can be the same capabilities that are defined for the **SrcBlendCaps** member.

AlphaCmpCaps

Alpha-test comparison capabilities. This member can include the same capability flags defined for the **ZCmpCaps** member. If this member contains only the D3DPCMPCAPS_ALWAYS capability or only the D3DPCMPCAPS_NEVER capability, the driver does not support alpha tests. Otherwise, the flags identify the individual comparisons that are supported for alpha testing.

ShadeCaps

Shading operations capabilities. It is assumed, in general, that if a device supports a given command at all, it supports the D3DSHADE_FLAT mode (as specified in the **D3DSHADEMODE** enumerated type). This flag specifies whether the driver can also support Gouraud shading and whether alpha color components are supported. When alpha components are not supported, the alpha value of colors generated is implicitly 255. This is the maximum possible alpha (that is, the alpha component is at full intensity).

The color, specular highlights, fog, and alpha interpolants of a triangle each have capability flags that an application can use to find out how they are implemented by the device driver.

This member can be one or more of the following flags.

D3DPSHADECAPS_ALPHAGOURAUDBLEND
Device can support an alpha component for Gouraud-blended transparency (the D3DSHADE_GOURAUD state for the **D3DSHADEMODE** enumerated type). In this mode, the alpha color component of a primitive is provided at vertices and interpolated across a face along with the other color components.

D3DPSHADECAPS_COLORGOURAUDRGB
Device can support colored Gouraud shading in the RGB color model. In this mode, the color component for a primitive is provided at vertices and interpolated across a face along with the other color components. In the RGB lighting model, the red, green, and blue components are interpolated.

D3DPSHADECAPS_FOGGOURAUD
Device can support fog in the Gouraud shading mode.

D3DPSHADECAPS_SPECULARGOURAUDRGB
Device can support specular highlights in Gouraud shading in the RGB color model.

TextureCaps

Miscellaneous texture-mapping capabilities. This member can be one or more of the following flags.

D3DPTEXTURECAPS_ALPHA
Alpha in texture pixels is supported.

D3DPTEXTURECAPS_ALPHAPALETTE
Device can draw alpha from texture palettes.

D3DPTEXTURECAPS_CUBEMAP
Supports cube textures

D3DPTEXTURECAPS_CUBEMAP_POW2
Device requires that cube texture maps have dimensions specified as powers of 2.

D3DPTEXTURECAPS_MIPCUBEMAP
Device supports mipmapped cube textures.

D3DPTEXTURECAPS_MIPMAP
Device supports mipmapped textures.

D3DPTEXTURECAPS_MIPVOLUMEMAP
Device supports mipmapped volume textures.

D3DPTEXTURECAPS_NONPOW2CONDITIONAL
Conditionally supports the use of textures with dimensions that are not powers of 2. A device that exposes this capability can use such a texture if all of the following requirements are met.

- The texture addressing mode for the texture stage is set to D3DTADDRESS_CLAMP.
- Texture wrapping for the texture stage is disabled (D3DRS_WRAP*n* set to 0).
- Mipmapping is not in use (use magnification filter only).

D3DPTEXTURECAPS_PERSPECTIVE
Perspective correction texturing is supported.

D3DPTEXTURECAPS_POW2
All textures must have widths and heights specified as powers of 2. This requirement does not apply to either cube textures or volume textures.

D3DPTEXTURECAPS_PROJECTED
Supports the D3DTTFF_PROJECTED texture transformation flag. When applied, the device divides transformed texture coordinates by the last texture coordinate. If this capability is present, then the projective divide occurs per pixel. If this capability is not present, but the projective divide needs to occur anyway, then it is performed on a per-vertex basis by the Direct3D runtime.

D3DPTEXTURECAPS_SQUAREONLY
All textures must be square.

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE
Texture indices are not scaled by the texture size prior to interpolation.

D3DPTEXTURECAPS_VOLUMEMAP
Device supports volume textures.

D3DPTEXTURECAPS_VOLUMEMAP_POW2
Device requires that volume texture maps have dimensions specified as powers of 2.

TextureFilterCaps
Texture-filtering capabilities for a Direct3DTexture object. Per-stage filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple-texture blending with the **IDirect3DDevice8** interface. This member can be any combination of the following per-stage texture-filtering flags.

D3DPTFILTERCAPS_MAGFAFLATCUBIC

Device supports per-stage flat cubic filtering for magnifying textures. The flat cubic magnification filter is represented by the D3DTEXF_FLATCUBIC member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MAGFANISOTROPIC
Device supports per-stage anisotropic filtering for magnifying textures. The anisotropic magnification filter is represented by the D3DTEXF_ANISOTROPIC member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC
Device supports the per-stage Gaussian cubic filtering for magnifying textures. The Gaussian cubic magnification filter is represented by the D3DTEXF_GAUSSIANCUBIC member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MAGFLINEAR
Device supports per-stage bilinear interpolation filtering for magnifying textures. The bilinear interpolation magnification filter is represented by the D3DTEXF_LINEAR member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MAGFPOINT
Device supports per-stage point-sample filtering for magnifying textures. The point-sample magnification filter is represented by the D3DTEXF_POINT member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MINFANISOTROPIC
Device supports per-stage anisotropic filtering for minifying textures. The anisotropic minification filter is represented by the D3DTEXF_ANISOTROPIC member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MINFLINEAR
Device supports per-stage bilinear interpolation filtering for minifying textures. The bilinear minification filter is represented by the D3DTEXF_LINEAR member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MINFPOINT
Device supports per-stage point-sample filtering for minifying textures. The point-sample minification filter is represented by the D3DTEXF_POINT member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MIPFLINEAR
Device supports per-stage trilinear interpolation filtering for mipmaps. The trilinear interpolation mipmapping filter is represented by the D3DTEXF_LINEAR member of the **D3DTEXTUREFILTERTYPE** enumerated type.

D3DPTFILTERCAPS_MIPFPOINT
Device supports per-stage point-sample filtering for mipmaps. The point-sample mipmapping filter is represented by the D3DTEXF_POINT member of the **D3DTEXTUREFILTERTYPE** enumerated type.

CubeTextureFilterCaps
Texture-filtering capabilities for a Direct3DCubeTexture object. Per-stage filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple-texture blending with the **IDirect3DDevice8** interface. This member can be any combination of the per-stage texture-filtering flags defined for the **TextureFilterCaps** member.

VolumeTextureFilterCaps
Texture-filtering capabilities for a Direct3DVolumeTexture object. Per-stage filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple-texture blending with the **IDirect3DDevice8** interface. This member can be any combination of the per-stage texture-filtering flags defined for the **TextureFilterCaps** member.

TextureAddressCaps
Texture-addressing capabilities for Direct3DTexture objects. This member can be one or more of the following flags.

D3DPTADDRESSCAPS_BORDER
Device supports setting coordinates outside the range [0.0, 1.0] to the border color, as specified by the D3DTSS_BORDERCOLOR texture-stage state.

D3DPTADDRESSCAPS_CLAMP
Device can clamp textures to addresses.

D3DPTADDRESSCAPS_INDEPENDENTUV
Device can separate the texture-addressing modes of the u and v coordinates of the texture. This ability corresponds to the D3DTSS_ADDRESSU and D3DTSS_ADDRESSV render-state values.

D3DPTADDRESSCAPS_MIRROR
Device can mirror textures to addresses.

D3DPTADDRESSCAPS_MIRRORONCE
Device can take the absolute value of the texture coordinate (thus, mirroring around 0), and then clamp to the maximum value.

D3DPTADDRESSCAPS_WRAP
Device can wrap textures to addresses.

VolumeTextureAddressCaps
Texture-addressing capabilities for Direct3DVolumeTexture objects. This member can be one or more of the flags defined for the **TextureAddressCaps** member.

LineCaps
Defines the capabilities for line-drawing primitives.

D3DLINECAPS_ALPHACMP
Supports alpha-test comparisons.

D3DLINECAPS_BLEND
Supports source-blending.

D3DLINECAPS_FOG
Supports fog.

D3DLINECAPS_TEXTURE

Supports texture-mapping.

D3DLINECAPS_ZTEST
Supports z-buffer comparisons.

MaxTextureWidth and **MaxTextureHeight**
Maximum texture width and height for this device.

MaxVolumeExtent
Maximum volume extent.

MaxTextureRepeat
Full range of the integer bits of the post-normalized texture indices. If the D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE bit is set, the device defers scaling by the texture size until after the texture address mode is applied. If not set, the device scales the texture indices by the texture size (largest level of detail) prior to interpolation.

If D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE is set, then the exact number of times a texture can be wrapped is **MaxTextureRepeat**. If D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE is not set, then the exact number of times a texture can be wrapped is (**MaxTextureRepeat** * texture size). For example, the device has D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE set, **MaxTextureRepeat** is 32k, and the maximum texture size is 4k. The device uses 27 integer bits (plus 5 fraction bits in a 32 bit (signed) integer), and thus has enough to wrap a 4k texture 32k times (assuming the texture coordinates equally span the positive and negative texture coordinate range). Alternately, the device could not set D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE and set **MaxTextureRepeat** to 2**27.

MaxTextureAspectRatio
Maximum texture aspect ratio supported by the hardware, typically a power of 2.

MaxAnisotropy
Maximum valid value for the D3DTSS_MAXANISOTROPY texture-stage state.

MaxVertexW
Maximum W-based depth value that the device supports.

GuardBandLeft, **GuardBandTop**, **GuardBandRight**, and **GuardBandBottom**
Screen-space coordinates of the guard-band clipping region. Coordinates inside this rectangle but outside the viewport rectangle are automatically clipped.

ExtentsAdjust
Number of pixels to adjust the extents rectangle outward to accommodate anti-aliasing kernels.

StencilCaps
Flags specifying supported stencil-buffer operations. Stencil operations are assumed to be valid for all three stencil-buffer operation render states (D3DRS_STENCILFAIL, D3DRS_STENCILPASS, and D3DRS_STENCILFAILZFAIL).

D3DSTENCILCAPS_DECR
The **D3DSTENCILOP_DECR** operation is supported.

D3DSTENCILCAPS_DECRSAT
The **D3DSTENCILOP_DECRSAT** operation is supported.

D3DSTENCILCAPS_INCR
The **D3DSTENCILOP_INCR** operation is supported.

D3DSTENCILCAPS_INCRSAT
The **D3DSTENCILOP_INCRSAT** operation is supported.

D3DSTENCILCAPS_INVERT
The **D3DSTENCILOP_INVERT** operation is supported.

D3DSTENCILCAPS_KEEP
The **D3DSTENCILOP_KEEP** operation is supported.

D3DSTENCILCAPS_REPLACE
The **D3DSTENCILOP_REPLACE** operation is supported.

D3DSTENCILCAPS_ZERO
The **D3DSTENCILOP_ZERO** operation is supported.

For more information, see the **D3DSTENCILOP** enumerated type.

FVFCaps

Flexible vertex format capabilities.

D3DFVFCAPS_DONOTSTRIPELEMENTS
It is preferable that vertex elements not be stripped. That is, if the vertex format contains elements that are not used with the current render states, there is no need to regenerate the vertices. If this capability flag is not present, stripping extraneous elements from the vertex format provides better performance.

D3DFVFCAPS_PSIZE
The absence of D3DFVFCAPS_PSIZE indicates that the device does not support D3DFVF_PSIZE for pre-transformed vertices. In this case the base point size always comes fromD3DRS_POINTSIZE. This capability applies to fixed-function vertex processing in software only. D3DFVF_PSIZE is always supported when doing software vertex processing.

The output point size written by a vertex shader is always supported, and for vertex shaders any input can contribute to the output point size.

D3DFVF_PSIZE is always supported for post-transformed vertices.

D3DFVFCAPS_TEXCOORDCOUNTMASK
Masks the low **WORD** of **FVFCaps**. These bits, cast to the **WORD** data type, describe the total number of texture coordinate sets that the device can simultaneously use for multiple texture blending. (You can use up to eight texture coordinate sets for any vertex, but the device can blend using only the specified number of texture coordinate sets.)

TextureOpCaps

Combination of flags describing the texture operations supported by this device. The following flags are defined.

D3DTEXOPCAPS_ADD
The **D3DTOP_ADD** texture-blending operation is supported.

D3DTEXOPCAPS_ADDSIGNED
The **D3DTOP_ADDSIGNED** texture-blending operation is supported.

D3DTEXOPCAPS_ADDSIGNED2X
The **D3DTOP_ADDSIGNED2X** texture-blending operation is supported.

D3DTEXOPCAPS_ADDSMOOTH
The **D3DTOP_ADDSMOOTH** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDCURRENTALPHA
The **D3DTOP_BLENDCURRENTALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDDIFFUSEALPHA
The **D3DTOP_BLENDDIFFUSEALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDFACTORALPHA
The **D3DTOP_BLENDFACTORALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDTEXTUREALPHA
The **D3DTOP_BLENDTEXTUREALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDTEXTUREALPHAPM
The **D3DTOP_BLENDTEXTUREALPHAPM** texture-blending operation is supported.

D3DTEXOPCAPS_BUMPENVMAP
The **D3DTOP_BUMPENVMAP** texture-blending operation is supported.

D3DTEXOPCAPS_BUMPENVMAPLUMINANCE
The **D3DTOP_BUMPENVMAPLUMINANCE** texture-blending operation is supported.

D3DTEXOPCAPS_DISABLE
The **D3DTOP_DISABLE** texture-blending operation is supported.

D3DTEXOPCAPS_DOTPRODUCT3
The **D3DTOP_DOTPRODUCT3** texture-blending operation is supported.

D3DTEXOPCAPS_LERP
The **D3DTOP_LERP** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATE
The **D3DTOP_MODULATE** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATE2X
The **D3DTOP_MODULATE2X** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATE4X
The **D3DTOP_MODULATE4X** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR
The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA

The **D3DTOP_MODULATECOLOR_ADDALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR
The **D3DTOP_MODULATEINVALPHA_ADDCOLOR** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA
The **D3DTOP_MODULATEINVCOLOR_ADDALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_MULTIPLYADD
The **D3DTOP_MULTIPLYADD** texture-blending operation is supported.

D3DTEXOPCAPS_PREMODULATE
The **D3DTOP_PREMODULATE** texture-blending operation is supported.

D3DTEXOPCAPS_SELECTARG1
The **D3DTOP_SELECTARG1** texture-blending operation is supported.

D3DTEXOPCAPS_SELECTARG2
The **D3DTOP_SELECTARG2** texture-blending operation is supported.

D3DTEXOPCAPS_SUBTRACT
The **D3DTOP_SUBTRACT** texture-blending operation is supported.

MaxTextureBlendStages
Maximum number of texture-blending stages supported. This value is the number of blenders available. In the DirectX 8.0 programmable pixel pipeline, this should correspond to the number of instructions supported by pixel shaders on this particular implementation.

MaxSimultaneousTextures
Maximum number of textures that can be simultaneously bound to the texture blending stages. This value is the number of textures that can be used in a single pass. In the DirectX 8.0 programmable pixel pipeline, this indicates the number of texture registers supported by pixel shaders on this particular piece of hardware, and the number of texture declaration instructions that can be present.

VertexProcessingCaps
Vertex processing capabilities. For a given physical device, this capability might vary across Direct3D Device Objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

D3DVTXPCAPS_DIRECTIONALLIGHTS
Device supports directional lights.

D3DVTXPCAPS_LOCALVIEWER
Device supports local viewer.

D3DVTXPCAPS_MATERIALSOURCE7
Device supports selectable vertex color sources.

D3DVTXPCAPS_POSITIONALLIGHTS
Device supports positional lights (including point lights and spotlights).

D3DVTXPCAPS_TEXGEN
Device can generate texture coordinates.

D3DVTXPCAPS_TWEENING
Device supports vertex tweening.

D3DVTXPCAPS_NO_VSDT_UBYTE4
Device does not support the D3DVSDT_UBYTE4 vertex declaration type.

MaxActiveLights
Maximum number of lights that can be active simultaneously. For a given physical device, this capability might vary across Direct3D Device Objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

MaxUserClipPlanes
Maximum number of user-defined clipping planes supported. This member can range from 0 through D3DMAXUSERCLIPPLANES. For a given physical device, this capability may vary across Direct3D Device Objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

MaxVertexBlendMatrices
Maximum number of matrices that this device can apply when performing multimatrix vertex blending. For a given physical device, this capability may vary across Direct3D Device Objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

MaxVertexBlendMatrixIndex
DWORD value that specifies the maximum matrix index that can be indexed into using the per-vertex indices. The number of matrices is **MaxVertexBlendMatrixIndex** + 1, which is the size of the matrix palette. If normals are present in the vertex data that needs to be blended for lighting, then the number of matrices is half the number specified by this capability flag. If **MaxVertexBlendMatrixIndex** is set to zero, the driver does not support indexed vertex blending. If this value is not zero then the valid range of indices is zero through **MaxVertexBlendIndexedMatrices**.

A zero value for **MaxVertexBlendMatrixIndex** indicates that the driver does not support indexed matrices.

When software vertex processing is used, 256 matrices could be used for indexed vertex blending, with or without normal blending.

For a given physical device, this capability may vary across Direct3D Device Objects depending on the parameters supplied to **IDirect3D8::CreateDevice**.

MaxPointSize
Maximum size of a point primitive. If set to 1.0f then device does not support point size control. The range is greater than or equal to 1.0f.

MaxPrimitiveCount
Maximum number of primitives for each DrawPrimitive call.

MaxVertexIndex
Maximum size of indices supported for hardware vertex processing. It is possible to create 32-bit index buffers by specifying D3DFMT_INDEX32; however, you will not be able to render with the index buffer unless this value is greater than 0x0000FFFF.

MaxStreams

Maximum number of concurrent data streams for **IDirect3DDevice8::SetStreamSource**. The valid range is 1 to 16. Note that if this value is 0, then the driver is not a DirectX 8.0 driver.

MaxStreamStride
Maximum stride for **IDirect3DDevice8::SetStreamSource**.

VertexShaderVersion
Vertex shader version, indicating the level of vertex shader supported by the device. Only vertex shaders with version numbers equal to or less than this will succeed in calls to **IDirect3DDevice8::CreateVertexShader**. The level of shader is specified to **CreateVertexShader** as the first token in the vertex shader token stream.

- DirectX 7.0 functionality is 0
- DirectX 8.0 functionality is 01

The main version number is encoded in the second byte. The low byte contains a sub-version number.

MaxVertexShaderConst
Number of vertex shader constant registers.

PixelShaderVersion
Pixel shader version, indicating the level of pixel shader supported by the device. Only pixel shaders with version numbers equal to or less than this will succeed in calls to **IDirect3DDevice8::CreatePixelShader**.

- DirectX 8.0 functionality is 01

The main version number is encoded in the second byte. The low byte contains a sub-version number.

MaxPixelShaderValue
Maximum value of pixel shader arithmetic component. This value indicates the internal range of values supported for pixel color blending operations. Within the range that they report to, implementations must allow data pass through pixel processing unmodified (unclamped). Normally, the value of this member is an absolute value. For example, a 1.0 indicates that the range is -1.0 to 1, and an 8.0 indicates that the range is -8.0 to 8.0. Note that the value 0.0 indicates that no signed range is supported; therefore, the range is 0 to 1.0 as in DirectX 6.0 and 7.0

Remarks

The **MaxTextureBlendStages** and **MaxSimultaneousTextures** members might seem very similar, but they contain different information. The **MaxTextureBlendStages** member contains the total number of texture-blending stages supported by the current device, and the **MaxSimultaneousTextures** member describes how many of those stages can have textures bound to them by using the **IDirect3DDevice8::SetTexture** method.

When the driver fills this structure, it can set values for execute-buffer capabilities, even when the interface being used to retrieve the capabilities (such as **IDirect3DDevice8**) does not support execute buffers.

For systems that set the D3DCAPS2_NO2DDURING3DSCENE capability flag, performance problems may occur if you use a texture and then modify it during a scene. This is true on all hardware, but it is more severe on hardware that exposes the D3DCAPS2_NO2DDURING3DSCENE capability. If D3DCAPS2_NO2DDURING3DSCENE is present on the hardware, application-based texture management should ensure that no texture used in the current **BeginScene** and **EndScene** block is evicted unless absolutely necessary. In the case of extremely high texture usage within a scene, the results are undefined. This occurs when you modify a texture that you have used in the scene and there is no spare texture memory available. For such systems, the contents of the z buffer become invalid at **EndScene**. Applications should not call **IDirect3DDevice8::CopyRects** to or from the back buffer on this type of hardware inside a **BeginScene** and **EndScene** pair. In addition, applications should not try to access the z buffer if the D3DPRASTERCAPS_ZBUFFERLESSHSR capability flag is set. Finally, applications should not lock the back buffer or the z buffer inside a **BeginScene** and **EndScene** pair.

The following flags concerning mipmapped textures are not supported in DirectX 8.0.

- D3DPTFILTERCAPS_NEAREST
- D3DPTFILTERCAPS_LINEAR
- D3DPTFILTERCAPS_MIPNEAREST
- D3DPTFILTERCAPS_MIPLINEAR
- D3DPTFILTERCAPS_LINEARMIPNEAREST
- D3DPTFILTERCAPS_LINEARMIPLINEAR

Requirements

Header: Declared in D3d8caps.h.

See Also

IDirect3D8::GetDeviceCaps, **IDirect3DDevice8::GetDeviceCaps**

D3DCLIPSTATUS8

Describes the current clip status.

```
typedef struct _D3DCLIPSTATUS8 {
  DWORD ClipUnion;
  DWORD ClipIntersection;
} D3DCLIPSTATUS8;
```

Members

ClipUnion
Clip union flags that describe the current clip status. This member can be one or more of the following flags.

D3DCS_ALL
Combination of all clip flags.

D3DCS_BACK
All vertices are clipped by the back plane of the viewing frustum.

D3DCS_BOTTOM
All vertices are clipped by the bottom plane of the viewing frustum.

D3DCS_FRONT
All vertices are clipped by the front plane of the viewing frustum.

D3DCS_LEFT
All vertices are clipped by the left plane of the viewing frustum.

D3DCS_RIGHT
All vertices are clipped by the right plane of the viewing frustum.

D3DCS_TOP
All vertices are clipped by the top plane of the viewing frustum.

D3DCS_PLANE0 through D3DCS_PLANE5
Application-defined clipping planes.

ClipIntersection
Clip intersection flags that describe the current clip status. This member can take the same flags as **ClipUnion**.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetClipStatus, **IDirect3DDevice8::SetClipStatus**

D3DCOLORVALUE

Describes color values.

```
typedef struct _D3DCOLORVALUE {
    float r;
    float g;
    float b;
    float a;
} D3DCOLORVALUE;
```

Members

r, **g**, **b**, and **a**
Floating-point values specifying the red, green, blue, and alpha components of a color. These values generally are in the range from 0.0 through 1.0, with 0.0 being black.

Remarks

You can set the members of this structure to values outside the range of 0 through 1 to implement some unusual effects. Values greater than 1 produce strong lights that tend to wash out a scene. Negative values produce dark lights that actually remove light from a scene.

Requirements

Header: Declared in D3d8types.h.

See Also

Color Values for Lights and Materials

D3DDEVICE_CREATION_PARAMETERS

Describes the creation parameters for a device.

```
typedef struct _D3DDEVICE_CREATION_PARAMETERS {
    UINT         AdapterOrdinal;
    D3DDEVTYPE   DeviceType;
    DWORD        hFocusWindow;
    HWND         BehaviorFlags;
} D3DDEVICE_CREATION_PARAMETERS;
```

Members

AdapterOrdinal
Ordinal number that denotes the display adapter. D3DADAPTER_DEFAULT is always the primary display adapter.

Use this ordinal as the *Adapter* parameter for any of the **IDirect3D8** methods. Note that different instances of Direct3D8 objects may use different ordinals. For example, adapters can enter and leave a system due to users adding or subtracting monitors from a multiple monitor system, or due to hot-swapping a laptop. Consequently, you should use this ordinal only in a **Direct3D8** known to be valid. The only two valid **Direct3D8** instances are the **Direct3D8** that created this **IDirect3DDevice8** interface and the **Direct3D8** returned from

IDirect3DDevice8::GetDirect3D,as called through this **IDirect3DDevice8** interface.

DeviceType
Member of the **D3DDEVTYPE** enumerated type. Denotes the amount of emulated functionality for this device. The value of this parameter mirrors the value passed to the **IDirect3D8::CreateDevice** call that created this device.

hFocusWindow
Window handle to which focus belongs for this Microsoft® Direct3D® device. The value of this parameter mirrors the value passed to the **IDirect3D8::CreateDevice** call that created this device.

BehaviorFlags
A combination of one or more of the following flags that control global behaviors of the Microsoft® Direct3D® device.

D3DCREATE_FPU_PRESERVE
The calling application does not want Direct3D to modify the FPU state in ways that are visible to the application. In this mode, Direct3D saves and restores the FPU state every time it needs to modify the FPU state.

D3DCREATE_HARDWARE_VERTEXPROCESSING
Specifies hardware vertex processing.

D3DCREATE_MIXED_VERTEXPROCESSING
Specifies mixed (both software and hardware) vertex processing.

D3DCREATE_MULTITHREADED
Requests multithread-safe behavior. This causes Direct3D to take the global critical section more frequently.

D3DCREATE_PUREDEVICE
Specifies hardware rasterization, transform, lighting, and shading.

D3DCREATE_SOFTWARE_VERTEXPROCESSING
Specifies software vertex processing.

The value of this parameter mirrors the value passed to the **IDirect3D8::CreateDevice** call that created this device.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetCreationParameters, **IDirect3D8::CreateDevice**

D3DDISPLAYMODE

Describes the display mode.

```
typedef struct _D3DDISPLAYMODE {
```

```
    UINT        Width;
    UINT        Height;
    UINT        RefreshRate;
    D3DFORMAT   Format;
} D3DDISPLAYMODE;
```

Members

Width
Screen width, in pixels.

Height
Screen height, in pixels.

RefreshRate
Refresh rate. The value of 0 indicates an adapter default.

Format
Member of the **D3DFORMAT** enumerated type, describing the surface format of the display mode.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3D8::EnumAdapterModes, **IDirect3D8::GetAdapterDisplayMode**, **IDirect3DDevice8::GetDisplayMode**

D3DGAMMARAMP

Contains red, green, and blue ramp data.

```
typedef struct _D3DGAMMARAMP {
    WORD        red  [256];
    WORD        green[256];
    WORD        blue [256];
} D3DGAMMARAMP;
```

Members

red, **green**, and **blue**
Array of 256 **WORD** elements that describe the red, green, and blue gamma ramps.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetGammaRamp, **IDirect3DDevice8::SetGammaRamp**

D3DINDEXBUFFER_DESC

Describes an index buffer.

```
typedef struct _D3DINDEXBUFFER_DESC {
    D3DFORMAT           Format;
    D3DRESOURCETYPE     Type;
    DWORD               Usage;
    D3DPOOL             Pool;
    UINT                Size;
} D3DINDEXBUFFER_DESC;
```

Members

Format
Member of the **D3DFORMAT** enumerated type, describing the surface format of the index buffer data.

Type
Member of the **D3DRESOURCETYPE** enumerated type, identifying this resource as an index buffer.

Usage
Combination of one or more of the following flags, specifying the usage for this resource.

D3DUSAGE_DONOTCLIP
Set to indicate that the index buffer content will never require clipping.

D3DUSAGE_RTPATCHES
Set to indicate when the index buffer is to be used for drawing high-order primitives.

D3DUSAGE_NPATCHES
Set to indicate when the index buffer is to be used for drawing N patches.

D3DUSAGE_POINTS
Set to indicate when the index buffer is to be used for drawing point sprites or indexed point lists.

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the buffer is to be used with software processing.

D3DUSAGE_WRITEONLY

Informs the system that the application writes only to the index buffer. Using this flag enables the driver to choose the best memory location for efficient write operations and rendering. Attempts to read from an index buffer that is created with this capability can result in degraded performance.

Pool

Member of the **D3DPOOL** enumerated type, specifying the class of memory allocated for this index buffer.

Size

Size of the index buffer, in bytes.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DIndexBuffer8::GetDesc

D3DLIGHT8

Defines a set of lighting properties.

```
typedef struct _D3DLIGHT8 {
    D3DLIGHTTYPE    Type;
    D3DCOLORVALUE   Diffuse;
    D3DCOLORVALUE   Specular;
    D3DCOLORVALUE   Ambient;
    D3DVECTOR       Position;
    D3DVECTOR       Direction;
    float           Range;
    float           Falloff;
    float           Attenuation0;
    float           Attenuation1;
    float           Attenuation2;
    float           Theta;
    float           Phi;
} D3DLIGHT8;
```

Members

Type

Type of the light source. This value is one of the members of the **D3DLIGHTTYPE** enumerated type.

Diffuse

Diffuse color emitted by the light. This member is a **D3DCOLORVALUE** structure.

Specular
Specular color emitted by the light. This member is a **D3DCOLORVALUE** structure.

Ambient
Ambient color emitted by the light. This member is a **D3DCOLORVALUE** structure.

Position
Position of the light in world space, specified by a **D3DVECTOR** structure. This member has no meaning for directional lights and is ignored in that case.

Direction
Direction that the light is pointing in world space, specified by a **D3DVECTOR** structure. This member only has meaning only for directional and spotlights. This vector need not be normalized, but it should have a nonzero length.

Range
Distance beyond which the light has no effect. The maximum allowable value for this member is the square root of FLT_MAX. This member does not affect directional lights.

Falloff
Decrease in illumination between a spotlight's inner cone (the angle specified by **Theta**) and the outer edge of the outer cone (the angle specified by **Phi**).

The effect of falloff on the lighting is subtle. Furthermore, a small performance penalty is incurred by shaping the falloff curve. For these reasons, most developers set this value to 1.0.

Attenuation0, **Attenuation1**, and **Attenuation2**
Values specifying how the light intensity changes over distance. Attenuation values are ignored for directional lights. These members represent attenuation constants. For information on attenuation, see Light Attenuation Over Distance. Valid values for these members range from 0.0 to infinity. For non-directional lights, all three attenuation values should not be set to 0.0 at the same time.

Theta
Angle, in radians, of a spotlight's inner cone—that is, the fully illuminated spotlight cone. This value must be in the range from 0 through the value specified by **Phi**.

Phi
Angle, in radians, defining the outer edge of the spotlight's outer cone. Points outside this cone are not lit by the spotlight. This value must be between 0 and *pi*.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetLight, **IDirect3DDevice8::SetLight**

D3DLINEPATTERN

Describes a line pattern.

```
typedef struct _D3DLINEPATTERN {
    WORD    wRepeatFactor;
    WORD    wLinePattern;
} D3DLINEPATTERN;
```

Members

wRepeatFactor
Number of times to repeat each series of 1s and 0s specified in the **wLinePattern** member. This allows an application to stretch the line pattern.

wLinePattern
Bits specifying the line pattern. For example, the following value would produce a dotted line: 1100110011001100.

Remarks

These values are used by the D3DRS_LINEPATTERN render state in the **D3DRENDERSTATETYPE** enumerated type.

A line pattern specifies how a line is drawn. The line pattern is always the same, no matter where it is started. (This differs from stippling, which affects how objects are rendered; that is, to imitate transparency.)

The line pattern specifies up to a 16-pixel pattern of on and off pixels along the line. The **wRepeatFactor** member specifies how many pixels are repeated for each entry in **wLinePattern**.

Requirements

Header: Declared in D3d8types.h.

D3DLOCKED_BOX

Describes a locked box (volume).

```
typedef struct _D3DLOCKED_BOX {
    INT             RowPitch;
    INT             SlicePitch;
    void*           pBits;
```

```
} D3DLOCKED_BOX;
```

Members

RowPitch
Byte offset from the left edge of one row to the left edge of the next row.

SlicePitch
Byte offset from the top-left of one slice to the top-left of the next deepest slice.

pBits
Pointer to the beginning of the volume box. If a D3DBOX was provided to the **LockRect** call, *pBits* will be appropriately offset from the start of the volume.

Remarks

Volumes can be visualized as being organized into slices of *width* x *height* 2-D surfaces stacked up to make a *width* x *height* x *depth* volume. For more information, see Volume Texture Resources.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DVolume8::LockBox, **IDirect3DVolumeTexture8::LockBox**

D3DLOCKED_RECT

Describes a locked rectangular region.

```
typedef struct _D3DLOCKED_RECT {
    INT             Pitch;
    void*           pBits;
} D3DLOCKED_RECT;
```

Members

Pitch
Pitch of surface, in bytes.

pBits
Pointer to the locked bits. If a RECT was provided to the **LockRect** call, *pBits* will be appropriately offset from the start of the surface.

Remarks

The pitch for DXTn formats is different from what was returned in DirectX 7.0. It now refers to a row of blocks. For example, if you have a width of 16, then you will have a pitch of 4 blocks (4*8 for DXT1, 4*16 for DXT2-5.)

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DCubeTexture8::LockRect, **IDirect3DSurface8::LockRect**, **IDirect3DTexture8::LockRect**.

D3DMATERIAL8

Specifies material properties.

```
typedef struct _D3DMATERIAL8 {
    D3DCOLORVALUE   Diffuse;
    D3DCOLORVALUE   Ambient;
    D3DCOLORVALUE   Specular;
    D3DCOLORVALUE   Emissive;
    float           Power;
} D3DMATERIAL8;
```

Members

Diffuse, **Ambient**, **Specular**, and **Emissive**
Values specifying the diffuse color, ambient color, specular color, and emissive color of the material, respectively. These values are **D3DCOLORVALUE** structures.

Power
Floating-point value specifying the sharpness of specular highlights. To turn off specular highlights for a material, set this member to 0.0; setting the specular color components to 0 is not enough.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetMaterial, **IDirect3DDevice8::SetMaterial**

D3DMATRIX

Describes a matrix.

```
typedef struct _D3DMATRIX {
    union {
        struct {
            float       _11, _12, _13, _14;
            float       _21, _22, _23, _24;
            float       _31, _32, _33, _34;
            float       _41, _42, _43, _44;

        };
        float m[4][4];
    };
} D3DMATRIX;
```

Remarks

In Microsoft® Direct3D®, the **_34** element of a projection matrix cannot be a negative number. If your application needs to use a negative value in this location, it should scale the entire projection matrix by –1, instead.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetTransform, **IDirect3DDevice8::MultiplyTransform**, **IDirect3DDevice8::SetTransform**

D3DPRESENT_PARAMETERS

Describes the presentation parameters.

```
typedef struct _D3DPRESENT_PARAMETERS_ {
    UINT                BackBufferWidth;
    UINT                BackBufferHeight;
    D3DFORMAT               BackBufferFormat;
    UINT                BackBufferCount;

    D3DMULTISAMPLE_TYPE     MultiSampleType;

    D3DSWAPEFFECT           SwapEffect;
    HWND                hDeviceWindow;
```

```
    BOOL            Windowed;
    BOOL            EnableAutoDepthStencil;
    D3DFORMAT           AutoDepthStencilFormat;
    DWORD           Flags;

    UINT            FullScreen_RefreshRateInHz;
    UINT            FullScreen_PresentationInterval;

} D3DPRESENT_PARAMETERS;
```

Members

BackBufferWidth and **BackBufferHeight**

Width and height of the new swap chain's back buffers, in pixels. If **Windowed** is FALSE (the presentation is full-screen), then these values must equal the width and height of one of the enumerated display modes found through **IDirect3D8::EnumAdapterModes**. If **Windowed** is TRUE and either of these values is zero, then the corresponding dimension of the client area of the **hDeviceWindow** (or the focus window, if **hDeviceWindow** is NULL) is taken.

BackBufferFormat

Member of the **D3DFORMAT** enumerated type. This value must be one of the render target formats as validated by **IDirect3D8::CheckDeviceType**.

If **Windowed** is set to TRUE, then **BackBufferFormat** must be set to match the format of the current display mode. Use **IDirect3DDevice8::GetDisplayMode** to obtain the current format.

BackBufferCount

This value can be 0, 1, 2 or 3. Note that 0 is treated as 1. If the number of back buffers cannot be created, the run time will fail the method call and fill this value with the number of back buffers that could be created. As a result, an application can call the method twice with the same **D3DPRESENT_PARAMETERS** structure and expect it to work the second time.

One back buffer is considered the minimum number of back buffers. The method call fails if 1 back buffer cannot be created. The value of **BackBufferCount** influences what set of swap effects are allowed. Specifically, any D3DSWAPEFFECT_COPY swap effect requires that there be exactly one back buffer.

MultiSampleType

Member of the **D3DMULTISAMPLE_TYPE** enumerated type. The value must be D3DMULTISAMPLE_NONE unless **SwapEffect** has been set to D3DSWAPEFFECT_DISCARD. Multisampling is supported only if the swap effect is D3DSWAPEFFECT_DISCARD.

SwapEffect

Member of the **D3DSWAPEFFECT** enumerated type. The run time will guarantee the implied semantics concerning buffer swap behavior. So if **Windowed** is TRUE and **SwapEffect** is set to D3DSWAPEFFECT_FLIP, then

the run time will create one extra back buffer, and copy whichever becomes the front buffer at presentation time.

D3DSWAPEFFECT_COPY and D3DSWAPEFFECT_COPY_VSYNC require that **BackBufferCount** be set to 1.

D3DSWAPEFFECT_DISCARD will be enforced in the debug run time by filling any buffer with noise after it is presented.

hDeviceWindow

If full-screen, this is the cover window. If windowed, this will be the default target window for **IDirect3DDevice8::Present**. If this value is NULL, the focus window will be taken. For applications that use multiple full-screen devices, such as a multimonitor system, exactly one device should use the focus window as the device window. All other devices should have unique device windows. Otherwise, behavior is undefined and applications will not work as expected.

Note that no attempt is made by the run time to reflect user changes in window size. The back buffer is not implicitly reset when this window is reset. However, the **Present** method does automatically track window position changes.

Windowed

TRUE if the application runs windowed, FALSE if the application runs full-screen.

EnableAutoDepthStencil

If this value is TRUE, Microsoft® Direct3D® will manage depth buffers for the application. The device will create a depth-stencil buffer when it is created. The depth-stencil buffer will be automatically set as the render target of the device. When the device is reset, the depth-stencil buffer will be automatically destroyed and recreated in the new size.

If **EnableAutoDepthStencil** is TRUE, then **AutoDepthStencilFormat** must be a valid depth-stencil format.

AutoDepthStencilFormat

Member of the **D3DFORMAT** enumerated type. The format of the automatic depth-stencil surface that the device will create. This member is ignored unless **EnableAutoDepthStencil** is TRUE.

Flags

This member can be set to 0, or to the following flag.

D3DPRESENTFLAG_LOCKABLE_BACKBUFFER

Set this flag if the application requires the ability to lock the back-buffer directly. Note that back buffers are not lockable unless the application specifies D3DPRESENTFLAG_LOCKABLE_BACKBUFFER when calling **IDirect3D8::CreateDevice** or **IDirect3DDevice8::Reset** . Lockable back buffers incur a performance cost on some graphics hardware configurations.

Performing a lock operation (or using **IDirect3DDevice8::CopyRects** to read/write) on the lockable back-buffer decreases performance on many cards. In this case, consider using textured triangles to move data to the back buffer.

FullScreen_RefreshRateInHz

The rate at which the display adapter refreshes the screen. For windowed mode, this value must be 0. Otherwise, this value must be one of the refresh rates returned by **IDirect3D8::EnumAdapterModes** or one of the following values.

D3DPRESENT_RATE_DEFAULT
The run time chooses the presentation rate, or adopts the current rate if windowed.

D3DPRESENT_RATE_UNLIMITED
The presentation rate runs as quickly as the hardware can deliver frames.

FullScreen_PresentationInterval
Maximum rate at which the swap chain's back buffers may be presented. For a windowed swap chain, this value must be D3DPRESENT_INTERVAL_DEFAULT (0). For a full-screen swap chain it may be D3DPRESENT_INTERVAL_DEFAULT or the value corresponding to exactly one of the flags enumerated in the *PresentationIntervals* member of **D3DCAPS8**.

D3DPRESENT_INTERVAL_IMMEDIATE
Present operations might be affected immediately. The driver will not wait for the vertical retrace period.

D3DPRESENT_INTERVAL_ONE
The driver will wait for the vertical retrace period. Present operations will not be affected more frequently than the screen refresh.

D3DPRESENT_INTERVAL_TWO
The driver will wait for the vertical retrace period. Present operations will not be affected more frequently than every second screen refresh.

D3DPRESENT_INTERVAL_THREE
The driver will wait for the vertical retrace period. Present operations will not be affected more frequently than every third screen refresh.

D3DPRESENT_INTERVAL_FOUR
The driver will wait for the vertical retrace period. Present operations will not be affected more frequently than every fourth screen refresh.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3D8::CreateDevice, **IDirect3DDevice8::CreateAdditionalSwapChain**, **IDirect3DDevice8::Present**, **IDirect3DDevice8::Reset**

D3DRANGE

Defines a range.

```
typedef struct _D3DRANGE {
    UINT        Offset;
```

```
    UINT            Size;
} D3DRANGE;
```

Members

Offset
Offset, in bytes.

Size
Size, in bytes.

Requirements

Header: Declared in D3d8types.h.

D3DRASTER_STATUS

Describes the raster status.

```
typedef struct _D3DRASTER_STATUS {
    BOOL        InVBlank;
    UINT        ScanLine;
} D3DRASTER_STATUS;
```

Members

InVBlank
TRUE if the raster is in the vertical blank period. FALSE if the raster is not in the vertical blank period.

ScanLine
If **InVBlank** is FALSE, then this value is an integer roughly corresponding to the current scan line painted by the raster. Scan lines are numbered in the same way as Microsoft® Direct3D® surface coordinates: 0 is the top of the primary surface, extending to the value (height of the surface - 1) at the bottom of the display.

If **InVBlank** is TRUE, then this value is set to zero and can be ignored.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetRasterStatus

D3DRECT

Defines a rectangle.

```
typedef struct _D3DRECT {
    LONG x1;
    LONG y1;
    LONG x2;
    LONG y2;
} D3DRECT;
```

Members

x1 and **y1**
Coordinates of the upper-left corner of the rectangle.

x2 and **y2**
Coordinates of the lower-right corner of the rectangle.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::Clear

D3DRECTPATCH_INFO

Describes a rectangular high-order patch.

```
typedef struct _D3DRECTPATCH_INFO {
    UINT            StartVertexOffsetWidth;
    UINT            StartVertexOffsetHeight;
    UINT            Width;
    UINT            Height;
    UINT            Stride;
    D3DBASISTYPE    Basis;
    D3DORDERTYPE    Order;
} D3DRECTPATCH_INFO;
```

Members

StartVertexOffsetWidth
Starting vertex offset width, in number of vertices.

StartVertexOffsetHeight

Starting vertex offset height, in number of vertices.

Width
Width of each vertex, in number of vertices.

Height
Height of each vertex, in number of vertices.

Stride
Stride between segments, in number of vertices.

Basis
Member of the **D3DBASISTYPE** enumerated type, defining the basis type for the rectangular high-order patch.

Order
Member of the **D3DORDERTYPE** enumerated type, defining the order type for the rectangular high-order patch.

Remarks

To render a stream of individual rectangular patches (non-mosaic), you should interpret your geometry as a long narrow (1 × N) rectangular patch. The **D3DRECTPATCH_INFO** structure for such a strip (cubic bézier) would be set up in the following manner.

```
D3DRECTPATCH_INFO RectInfo;

RectInfo.Width = 4;
RectInfo.Height= 4;
RectInfo.Stride= 4;
RectInfo.Basis =  D3DBASIS_BEZIER;
RectInfo.Order =  D3DORDER_CUBIC;
RectInfo.StartVertexOffsetWidth = 0;
RectInfo.StartVertexOffsetHeight = 4*i;  // The variable i is the index of the patch you want to
render.
```

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::DrawRectPatch

D3DSURFACE_DESC

Describes a surface.

```
typedef struct _D3DSURFACE_DESC {
```

```
    D3DFORMAT         Format;
    D3DRESOURCETYPE   Type;
    DWORD             Usage;
    D3DPOOL           Pool;
    UINT              Size;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    UINT              Width;
    UINT              Height;
} D3DSURFACE_DESC;
```

Members

Format
Member of the **D3DFORMAT** enumerated type, describing the surface format.

Type
Member of the **D3DRESOURCETYPE** enumerated type, identifying this resource as a surface.

Usage
Combination of one or more of the following flags, specifying the usage for this resource.

D3DUSAGE_DEPTHSTENCIL
Set to indicate that the surface is to be used as a depth stencil surface.

D3DUSAGE_RENDERTARGET
Set to indicate that the surface is to be used as a render target.

Pool
Member of the **D3DPOOL** enumerated type, specifying the class of memory allocated for this surface.

Size
Size of the surface, in bytes.

MultiSampleType
Member of the **D3DMULTISAMPLE_TYPE** enumerated type, specifying the levels of full-scene multisampling supported by the surface.

Width
Width of the surface, in pixels.

Height
Height of the surface, in pixels.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DCubeTexture8::GetLevelDesc, **IDirect3DSurface8::GetDesc**, **IDirect3DTexture8::GetLevelDesc**.

D3DTRIPATCH_INFO

Describes a triangular high-order patch.

```
typedef struct _D3DTRIPATCH_INFO {
    UINT            StartVertexOffset;
    UINT            NumVertices;
    D3DBASISTYPE    Basis;
    D3DORDERTYPE    Order;
} D3DTRIPATCH_INFO;
```

Members

StartVertexOffset
Starting vertex offset, in number of vertices.

NumVertices
Number of vertices.

Basis
Member of the **D3DBASISTYPE** enumerated type, defining the basis type for the triangular high-order patch.

Order
Member of the **D3DORDERTYPE** enumerated type, defining the order type for the triangular high-order patch.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::DrawTriPatch

D3DVECTOR

Defines a vector.

```
typedef struct _D3DVECTOR {
    float x;
    float y;
    float z;
} D3DVECTOR;
```

Members

x, **y** and **z**
Floating-point values describing the vector.

Requirements

Header: Declared in D3d8types.h.

D3DVERTEXBUFFER_DESC

Describes a vertex buffer.

```
typedef struct _D3DVERTEXBUFFER_DESC {
    D3DFORMAT           Format;
    D3DRESOURCETYPE     Type;
    DWORD               Usage;
    D3DPOOL             Pool;
    UINT                Size;
    DWORD               FVF;
} D3DVERTEXBUFFER_DESC;
```

Members

Format
Member of the **D3DFORMAT** enumerated type, describing the surface format of the vertex buffer data.

Type
Member of the **D3DRESOURCETYPE** enumerated type, identifying this resource as a vertex buffer.

Usage
Combination of one or more of the following flags, specifying the usage for this resource.

D3DUSAGE_DONOTCLIP
Set to indicate that the vertex buffer content will never require clipping.

D3DUSAGE_RTPATCHES
Set to indicate when the vertex buffer is to be used for drawing high-order primitives.

D3DUSAGE_NPATCHES
Set to indicate when the vertex buffer is to be used for drawing N patches.

D3DUSAGE_POINTS
Set to indicate when the vertex buffer is to be used for drawing point sprites or indexed point lists.

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the vertex buffer is to be used with software vertex processing.

D3DUSAGE_WRITEONLY
Informs the system that the application writes only to the vertex buffer. Using this flag enables the driver to choose the best memory location for efficient write operations and rendering. Attempts to read from a vertex buffer that is created with this capability can result in degraded performance.

Pool
Member of the **D3DPOOL** enumerated type, specifying the class of memory allocated for this vertex buffer.

Size
Size of the vertex buffer, in bytes

FVF
Combination of flexible vertex format flags that describes the vertex format of the vertices in this buffer.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DVertexBuffer8::GetDesc

D3DVIEWPORT8

Defines the window dimensions of a render target surface onto which a 3-D volume projects.

```
typedef struct _D3DVIEWPORT8 {
    DWORD       X;
    DWORD       Y;
    DWORD       Width;
    DWORD       Height;
    float       MinZ;
    float       MaxZ;
} D3DVIEWPORT8;
```

Members

X and **Y**
Pixel coordinates of the upper-left corner of the viewport on the render target surface. Unless you want to render to a subset of the surface, these members can be set to 0.

Width and **Height**
Dimensions of the viewport on the render target surface, in pixels. Unless you are rendering only to a subset of the surface, these members should be set to the dimensions of the render target surface.

MinZ and **MaxZ**
Values describing the range of depth values into which a scene is to be rendered, the minimum and maximum values of the clip volume. Most applications set these values to 0.0 and 1.0, respectively. Clipping is performed after applying the projection matrix.

Remarks

The **X**, **Y**, **Width**, and **Height** members describe the position and dimensions of the viewport on the render target surface. Usually, applications render to the entire target surface; when rendering on a 640×480 surface, these members should be 0, 0, 640, and 480, respectively. The **MinZ** and **MaxZ** are typically set to 0.0 and 1.0 but can be set to other values to achieve specific effects. For example, you might set them both to 0.0 to force the system to render objects to the foreground of a scene, or both to 1.0 to force the objects into the background.

When the viewport parameters for a device change (due to a call to the **IDirect3DDevice8::SetViewport** method), the driver builds a new transformation matrix.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetViewport, **IDirect3DDevice8::SetViewport**

D3DVOLUME_DESC

Describes a volume.

```
typedef struct _D3DVOLUME_DESC {
    D3DFORMAT         Format;
    D3DRESOURCETYPE   Type;
    DWORD             Usage;
    D3DPOOL           Pool;
    UINT              Size;
    UINT              Width;
    UINT              Height;
    UINT              Depth;
} D3DVOLUME_DESC;
```

Members

Format
Member of the **D3DFORMAT** enumerated type, describing the surface format of the volume.

Type
Member of the **D3DRESOURCETYPE** enumerated type, identifying this resource as a volume.

Usage
Currently not used. Always returned as 0.

Pool
Member of the **D3DPOOL** enumerated type, specifying the class of memory allocated for this volume.

Size
Size of the volume, in bytes.

Width
Width of the volume, in pixels.

Height
Height of the volume, in pixels.

Depth
Depth of the volume, in pixels.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DVolume8::GetDesc, **IDirect3DVolumeTexture8::GetLevelDesc**

Enumerated Types

This section contains information about the following enumerated types used with Microsoft® Direct3D®.

- **D3DBACKBUFFER_TYPE**
- **D3DBLEND**
- **D3DBLENDOP**
- **D3DCMPFUNC**
- **D3DCUBEMAP_FACES**
- **D3DCULL**
- **D3DDEBUGMONITORTOKENS**

- **D3DDEVTYPE**
- **D3DFILLMODE**
- **D3DFOGMODE**
- **D3DFORMAT**
- **D3DBASISTYPE**
- **D3DLIGHTTYPE**
- **D3DMATERIALCOLORSOURCE**
- **D3DMULTISAMPLE_TYPE**
- **D3DORDERTYPE**
- **D3DPATCHEDGESTYLE**
- **D3DPOOL**
- **D3DPRIMITIVETYPE**
- **D3DRENDERSTATETYPE**
- **D3DRESOURCETYPE**
- **D3DSHADEMODE**
- **D3DSTATEBLOCKTYPE**
- **D3DSTENCILOP**
- **D3DSWAPEFFECT**
- **D3DTEXTUREADDRESS**
- **D3DTEXTUREFILTERTYPE**
- **D3DTEXTUREOP**
- **D3DTEXTURESTAGESTATETYPE**
- **D3DTEXTURETRANSFORMFLAGS**
- **D3DTRANSFORMSTATETYPE**
- **D3DVERTEXBLENDFLAGS**
- **D3DZBUFFERTYPE**

D3DBACKBUFFER_TYPE

Defines constants that describe the type of back buffer.

```
typedef enum _D3DBACKBUFFER_TYPE {
    D3DBACKBUFFER_TYPE_MONO        = 0,
    D3DBACKBUFFER_TYPE_LEFT        = 1,
    D3DBACKBUFFER_TYPE_RIGHT       = 2,

    D3DBACKBUFFER_TYPE_FORCE_DWORD  = 0x7fffffff
} D3DBACKBUFFER_TYPE;
```

Constants

D3DBACKBUFFER_TYPE_MONO
Specifies a non-stereo swap chain.

D3DBACKBUFFER_TYPE_LEFT
Specifies the left side of a stereo pair in a swap chain.

D3DBACKBUFFER_TYPE_RIGHT
Specifies the right side of a stereo pair in a swap chain.

D3DBACKBUFFER_TYPE_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Note that stereo view in not supported in DirectX 8.0, so the D3DBACKBUFFER_TYPE_LEFT and D3DBACKBUFFER_TYPE_RIGHT values of this enumerated type are not used by Direct3D.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetBackBuffer, **IDirect3DSwapChain8::GetBackBuffer**

D3DBLEND

Defines the supported blend mode.

```
typedef enum _D3DBLEND {
    D3DBLEND_ZERO            =  1,
    D3DBLEND_ONE             =  2,
    D3DBLEND_SRCCOLOR        =  3,
    D3DBLEND_INVSRCCOLOR     =  4,
    D3DBLEND_SRCALPHA        =  5,
    D3DBLEND_INVSRCALPHA     =  6,
    D3DBLEND_DESTALPHA       =  7,
    D3DBLEND_INVDESTALPHA    =  8,
    D3DBLEND_DESTCOLOR       =  9,
    D3DBLEND_INVDESTCOLOR    = 10,
    D3DBLEND_SRCALPHASAT     = 11,
    D3DBLEND_BOTHSRCALPHA    = 12,
    D3DBLEND_BOTHINVSRCALPHA = 13,

    D3DBLEND_FORCE_DWORD     = 0x7fffffff
} D3DBLEND;
```

Constants

D3DBLEND_ZERO
Blend factor is (0, 0, 0, 0).

D3DBLEND_ONE
Blend factor is (1, 1, 1, 1).

D3DBLEND_SRCCOLOR
Blend factor is (R_s, G_s, B_s, A_s).

D3DBLEND_INVSRCCOLOR
Blend factor is (1–R_s, 1–G_s, 1–B_s, 1–A_s).

D3DBLEND_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DBLEND_INVSRCALPHA
Blend factor is (1–A_s, 1–A_s, 1–A_s, 1–A_s).

D3DBLEND_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DBLEND_INVDESTALPHA
Blend factor is (1–A_d, 1–A_d, 1–A_d, 1–A_d).

D3DBLEND_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DBLEND_INVDESTCOLOR
Blend factor is (1–R_d, 1–G_d, 1–B_d, 1–A_d).

D3DBLEND_SRCALPHASAT
Blend factor is (f, f, f, 1); f = min(A_s, 1–A_d).

D3DBLEND_BOTHSRCALPHA
Obsolete. For Microsoft® DirectX® 6.0 and later, you can achieve the same affect by setting the source and destination blend factors to D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA in separate calls.

D3DBLEND_BOTHINVSRCALPHA
Source blend factor is (1–A_s, 1–A_s, 1–A_s, 1–A_s), and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden. This blend mode is supported only for the D3DRS_SRCBLEND render state.

D3DBLEND_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

In the member descriptions above, the RGBA values of the source and destination are indicated by the subscripts *s* and *d*.

The values in this enumerated type are used by the D3DRS_DESTBLEND render state.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE

D3DBLENDOP

Defines the supported blend operations.

```
typedef enum _D3DBLENDOP {
    D3DBLENDOP_ADD           = 1,
    D3DBLENDOP_SUBTRACT      = 2,
    D3DBLENDOP_REVSUBTRACT   = 3,
    D3DBLENDOP_MIN           = 4,
    D3DBLENDOP_MAX           = 5,

    D3DBLENDOP_FORCE_DWORD   = 0x7fffffff
} D3DBLENDOP;
```

Constants

D3DBLENDOP_ADD
: The result is the destination added to the source.

  Result = Source + Destination

D3DBLENDOP_SUBTRACT
: The result is the destination subtracted from to the source.

  Result = Source - Destination

D3DBLENDOP_REVSUBTRACT
: The result is the source subtracted from the destination.

  Result = Destination - Source

D3DBLENDOP_MIN
: The result is the minimum of the source and destination.

  Result = MIN(Source, Destination)

D3DBLENDOP_MAX
: The result is the maximum of the source and destination.

  Result = MAX(Source, Destination)

D3DBLENDOP_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

This enumerated type defines values used by the D3DRS_BLENDOP render state.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DCAPS8, **D3DRENDERSTATETYPE**

D3DCMPFUNC

Defines the supported compare functions.

```
typedef enum _D3DCMPFUNC {
    D3DCMP_NEVER              = 1,
    D3DCMP_LESS               = 2,
    D3DCMP_EQUAL              = 3,
    D3DCMP_LESSEQUAL          = 4,
    D3DCMP_GREATER            = 5,
    D3DCMP_NOTEQUAL           = 6,
    D3DCMP_GREATEREQUAL       = 7,
    D3DCMP_ALWAYS             = 8,

    D3DCMP_FORCE_DWORD        = 0x7fffffff
} D3DCMPFUNC;
```

Constants

D3DCMP_NEVER
Always fail the test.

D3DCMP_LESS
Accept the new pixel if its value is less than the value of the current pixel.

D3DCMP_EQUAL
Accept the new pixel if its value equals the value of the current pixel.

D3DCMP_LESSEQUAL
Accept the new pixel if its value is less than or equal to the value of the current pixel.

D3DCMP_GREATER
Accept the new pixel if its value is greater than the value of the current pixel.

D3DCMP_NOTEQUAL
Accept the new pixel if its value does not equal the value of the current pixel.

D3DCMP_GREATEREQUAL

Accept the new pixel if its value is greater than or equal to the value of the current pixel.

D3DCMP_ALWAYS
Always pass the test.

D3DCMP_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The values in this enumerated type define the supported compare functions for the D3DRS_ZFUNC, D3DRS_ALPHAFUNC, and D3DRS_STENCILFUNC render states.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE

D3DCUBEMAP_FACES

Defines the faces of a cubemap.

```
typedef enum _D3DCUBEMAP_FACES {
    D3DCUBEMAP_FACE_POSITIVE_X     = 0,
    D3DCUBEMAP_FACE_NEGATIVE_X     = 1,
    D3DCUBEMAP_FACE_POSITIVE_Y     = 2,
    D3DCUBEMAP_FACE_NEGATIVE_Y     = 3,
    D3DCUBEMAP_FACE_POSITIVE_Z     = 4,
    D3DCUBEMAP_FACE_NEGATIVE_Z     = 5,

    D3DCUBEMAP_FACE_FORCE_DWORD    = 0xffffffff
} D3DCUBEMAP_FACES;
```

Constants

D3DCUBEMAP_FACE_POSITIVE_X
Positive x-face of the cubemap.

D3DCUBEMAP_FACE_NEGATIVE_X
Negative x-face of the cubemap.

D3DCUBEMAP_FACE_POSITIVE_Y
Positive y-face of the cubemap.

D3DCUBEMAP_FACE_NEGATIVE_Y

Negative y-face of the cubemap.

D3DCUBEMAP_FACE_POSITIVE_Z
Positive z-face of the cubemap.

D3DCUBEMAP_FACE_NEGATIVE_Z
Negative z-face of the cubemap.

D3DCUBEMAP_FACE_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DCubeTexture8::AddDirtyRect, **IDirect3DCubeTexture8::GetCubeMapSurface**, **IDirect3DCubeTexture8::LockRect**, **IDirect3DCubeTexture8::UnlockRect**

D3DCULL

Defines the supported culling modes.

```
typedef enum _D3DCULL {
    D3DCULL_NONE             = 1,
    D3DCULL_CW               = 2,
    D3DCULL_CCW              = 3,

    D3DCULL_FORCE_DWORD      = 0x7fffffff
} D3DCULL;
```

Constants

D3DCULL_NONE
Do not cull back faces.

D3DCULL_CW
Cull back faces with clockwise vertices.

D3DCULL_CCW
Cull back faces with counterclockwise vertices.

D3DCULL_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The values in this enumerated type are used by the D3DRS_CULLMODE render state. The culling modes define how back faces are culled when rendering a geometry.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DCAPS8, **D3DRENDERSTATETYPE**

D3DDEBUGMONITORTOKENS

Defines the debug monitor tokens.

```
typedef enum _D3DDEBUGMONITORTOKENS {
    D3DDMT_ENABLE         = 0,
    D3DDMT_DISABLE        = 1,

    D3DDMT_FORCE_DWORD    = 0x7fffffff
} D3DDEBUGMONITORTOKENS;
```

Constants

D3DDMT_ENABLE
Enable the debug monitor.

D3DDMT_DISABLE
Disable the debug monitor.

D3DDMT_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The values in this enumerated type are used by the D3DRS_DEBUGMONITORTOKEN render state and are only relevant for debug builds.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE

D3DDEVTYPE

Defines device types.

```
typedef enum _D3DDEVTYPE {
    D3DDEVTYPE_HAL         = 1,
    D3DDEVTYPE_REF         = 2,
    D3DDEVTYPE_SW          = 3,

    D3DDEVTYPE_FORCE_DWORD = 0xffffffff
} D3DDEVTYPE;
```

Constants

D3DDEVTYPE_HAL
: Hardware rasterization and shading with software, hardware, or mixed transform and lighting.

D3DDEVTYPE_REF
: Microsoft® Direct3D® features are implemented in software; however, the reference rasterizer does make use of special CPU instructions whenever it can.

D3DDEVTYPE_SW
: A pluggable software device that has been registered with Direct3D using **IDirect3D8::RegisterSoftwareDevice**.

D3DDEVTYPE_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3D8::CheckDeviceFormat, **IDirect3D8::CheckDeviceMultiSampleType**, **IDirect3D8::CheckDeviceType**, **IDirect3D8::CreateDevice**, **IDirect3D8::GetDeviceCaps**, **D3DDEVICE_CREATION_PARAMETERS**

D3DFILLMODE

Defines constants describing the fill mode.

```
typedef enum _D3DFILLMODE {
    D3DFILL_POINT          = 1,
```

```
    D3DFILL_WIREFRAME          = 2,
    D3DFILL_SOLID              = 3,

    D3DFILL_FORCE_DWORD        = 0x7fffffff
} D3DFILLMODE;
```

Constants

D3DFILL_POINT
Fill points.

D3DFILL_WIREFRAME
Fill wireframes. This fill mode currently does not work for clipped primitives when you use the DrawPrimitive methods.

D3DFILL_SOLID
Fill solids.

D3DFILL_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The values in this enumerated type are used by the D3DRS_FILLMODE render state.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE

D3DFOGMODE

Defines constants that describe the fog mode.

```
typedef enum _D3DFOGMODE {
    D3DFOG_NONE               = 0,
    D3DFOG_EXP                = 1,
    D3DFOG_EXP2               = 2,
    D3DFOG_LINEAR             = 3,

    D3DFOG_FORCE_DWORD        = 0x7fffffff
} D3DFOGMODE;
```

Constants

D3DFOG_NONE
No fog effect.

D3DFOG_EXP
Fog effect intensifies exponentially, according to the following formula.

$$f = 1 / e^{d \times density}$$

D3DFOG_EXP2
Fog effect intensifies exponentially with the square of the distance, according to the following formula.

$$f = 1 / e^{(d \times density)^2}$$

D3DFOG_LINEAR
Fog effect intensifies linearly between the start and end points, according to the following formula.

$$f = \frac{ena - a}{end - start}$$

This is the only fog mode currently supported.

D3DFOG_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The values in this enumerated type are used by the D3DRS_FOGTABLEMODE and D3DRS_FOGVERTEXMODE render states.

Fog can be considered a measure of visibility—the lower the fog value produced by a fog equation, the less visible an object is.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE

D3DFORMAT

Defines the various types of surface formats.

```
typedef enum _D3DFORMAT {
    D3DFMT_UNKNOWN          =  0,

    D3DFMT_R8G8B8           = 20,
    D3DFMT_A8R8G8B8         = 21,
    D3DFMT_X8R8G8B8         = 22,
    D3DFMT_R5G6B5           = 23,
    D3DFMT_X1R5G5B5         = 24,
    D3DFMT_A1R5G5B5         = 25,
    D3DFMT_A4R4G4B4         = 26,
    D3DFMT_R3G3B2           = 27,
    D3DFMT_A8               = 28,
    D3DFMT_A8R3G3B2         = 29,
    D3DFMT_X4R4G4B4         = 30,

    D3DFMT_A8P8             = 40,
    D3DFMT_P8               = 41,

    D3DFMT_L8               = 50,
    D3DFMT_A8L8             = 51,
    D3DFMT_A4L4             = 52,

    D3DFMT_V8U8             = 60,
    D3DFMT_L6V5U5           = 61,
    D3DFMT_X8L8V8U8         = 62,
    D3DFMT_Q8W8V8U8         = 63,
    D3DFMT_V16U16           = 64,
    D3DFMT_W11V11U10        = 65,

    D3DFMT_UYVY             = MAKEFOURCC('U', 'Y', 'V', 'Y'),
    D3DFMT_YUY2             = MAKEFOURCC('Y', 'U', 'Y', '2'),
    D3DFMT_DXT1             = MAKEFOURCC('D', 'X', 'T', '1'),
    D3DFMT_DXT2             = MAKEFOURCC('D', 'X', 'T', '2'),
    D3DFMT_DXT3             = MAKEFOURCC('D', 'X', 'T', '3'),
    D3DFMT_DXT4             = MAKEFOURCC('D', 'X', 'T', '4'),
    D3DFMT_DXT5             = MAKEFOURCC('D', 'X', 'T', '5'),

    D3DFMT_D16_LOCKABLE     = 70,
    D3DFMT_D32              = 71,
    D3DFMT_D15S1            = 73,
    D3DFMT_D24S8            = 75,
    D3DFMT_D16              = 80,
```

```
    D3DFMT_D24X8            =  77,
    D3DFMT_D24X4S4          =  79,

    D3DFMT_VERTEXDATA       = 100,
    D3DFMT_INDEX16          = 101,
    D3DFMT_INDEX32          = 102,

    D3DFMT_FORCE_DWORD      = 0xFFFFFFFF
} D3DFORMAT;
```

Constants

D3DFMT_UNKNOWN
Surface format is unknown.

D3DFMT_R8G8B8
24-bit RGB pixel format.

D3DFMT_A8R8G8B8
32-bit ARGB pixel format with alpha.

D3DFMT_X8R8G8B8
32-bit RGB pixel format where 8 bits are reserved for each color.

D3DFMT_R5G6B5
16-bit RGB pixel format.

D3DFMT_X1R5G5B5
16-bit pixel format where 5 bits are reserved for each color.

D3DFMT_A1R5G5B5
16-bit pixel format where 5 bits are reserved for each color and 1 bit is reserved for alpha (transparent texel).

D3DFMT_A4R4G4B4
16-bit ARGB pixel format.

D3DFMT_R3G3B2
8-bit RGB texture format.

D3DFMT_A8
8-bit alpha only.

D3DFMT_A8R3G3B2
16-bit ARGB texture format.

D3DFMT_X4R4G4B4
16-bit RGB pixel format where 4 bits are reserved for each color.

D3DFMT_A8P8
Surface is 8-bit color indexed with 8 bits of alpha.

D3DFMT_P8
Surface is 8-bit color indexed.

D3DFMT_L8
8-bit luminance only.

D3DFMT_A8L8
16-bit alpha luminance.

D3DFMT_A4L4
8-bit alpha luminance.

D3DFMT_V8U8
16-bit bump-map format.

D3DFMT_L6V5U5
16-bit bump-map format with luminance.

D3DFMT_X8L8V8U8
32-bit bump-map format with luminance where 8 bits are reserved for each element.

D3DFMT_Q8W8V8U8
32-bit bump-map format.

D3DFMT_V16U16
32-bit bump-map format.

D3DFMT_W11V11U10
32-bit bump-map format.

D3DFMT_UYVY
UYVY format (PC98 compliance).

D3DFMT_YUY2
YUY2 format (PC98 compliance).

D3DFMT_DXT1
DXT1 compression texture format.

D3DFMT_DXT2
DXT2 compression texture format.

D3DFMT_DXT3
DXT3 compression texture format.

D3DFMT_DXT4
DXT4 compression texture format.

D3DFMT_DXT5
DXT5 compression texture format.

D3DFMT_D16_LOCKABLE
16-bit z-buffer bit depth. This is an application-lockable surface format.

D3DFMT_D32
32-bit z-buffer bit depth.

D3DFMT_D15S1
16-bit z-buffer bit depth where 15 bits are reserved for the depth channel and 1 bit is reserved for the stencil channel.

D3DFMT_D24S8
32-bit z-buffer bit depth where 24 bits are reserved for the depth channel and 8 bits are reserved for the stencil channel.

D3DFMT_D16
16-bit z-buffer bit depth.

D3DFMT_D24X8
32-bit z-buffer bit depth where 24 bits are reserved for the depth channel.

D3DFMT_D24X4S4
32-bit z-buffer bit depth where 24 bits are reserved for the depth channel and 4 bits are reserved for the stencil channel.

D3DFMT_VERTEXDATA
Describes a vertex buffer surface.

D3DFMT_INDEX16
16-bit index buffer bit depth.

D3DFMT_INDEX32
32-bit index buffer bit depth.

D3DFMT_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Note that render target formats are restricted to D3DFMT_X1R5G5B5, D3DFMT_R5G6B5, D3DFMT_X8R8G8B8, and D3DFMT_A8R8G8B8.

The order of the bits are from the most significant bit (MSB) first, so D3DFMT_A8L8 indicates that the high byte of this two-byte format is alpha. D3DFMT_D16 indicates a 16-bit integer value and an application-lockable surface.

All depth-stencil formats except D3DFMT_D16_LOCKABLE indicate no particular bit ordering per pixel, and are not application-lockable, and the driver is allowed to consume more than the indicated number of bits per depth channel (but not stencil channel).

Pixel formats are denoted by opaque **DWORD** identifiers. The format of these **DWORD**s has been chosen to enable the expression of IHV-defined extension formats, and also to include the well-established FOURCC method. The set of formats understood by the Microsoft® Direct3D® runtime is defined by **D3DFORMAT**.

Note that IHV-supplied formats and many FOURCC codes are not listed in the **D3DFORMAT** enumeration. The formats in this enumeration are unique in that they are sanctioned by the runtime, meaning that the reference rasterizer will operate on all these types. The IHV-supplied formats will be supported by the individual IHVs on a card-by-card basis.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3D8::CheckDeviceFormat, Four-Character Codes (FOURCC)

D3DBASISTYPE

Defines the basis type of a high-order patch surface.

```
typedef enum _D3DBASISTYPE{
  D3DBASIS_BEZIER       = 0,
  D3DBASIS_BSPLINE      = 1,
  D3DBASIS_INTERPOLATE  = 2,

  D3DBASIS_FORCE_DWORD  = 0x7fffffff
} D3DBASISTYPE;
```

Constants

D3DBASIS_BEZIER
: Input vertices are treated as a series of bézier patches. The number of vertices specified must be divisible by 3 + 1. Portions of the mesh beyond this criterion will not be rendered. Full continuity is assumed between sub-patches in the interior of the surface rendered by each call. Only the vertices at the corners of each sub-patch are guaranteed to lie on the resulting surface.

D3DBASIS_BSPLINE
: Input vertices are treated as control points of a B-spline surface. The number of apertures rendered is 2 less than the number of apertures in that direction. In general, the generated surface does not contain the control vertices specified.

D3DBASIS_INTERPOLATE
: An interpolating basis defines the surface so that the surface goes through all the input vertices specified.

D3DBASIS_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The members of **D3DBASISTYPE** specify the formulation to be used in evaluating the high-order patch surface primitive during tessellation.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRECTPATCH_INFO, **D3DTRIPATCH_INFO**

D3DLIGHTTYPE

Defines the light type.

```
typedef enum _D3DLIGHTTYPE {
    D3DLIGHT_POINT          = 1,
    D3DLIGHT_SPOT           = 2,
    D3DLIGHT_DIRECTIONAL    = 3,

    D3DLIGHT_FORCE_DWORD    = 0x7fffffff
} D3DLIGHTTYPE;
```

Constants

D3DLIGHT_POINT
: Light is a point source. The light has a position in space and radiates light in all directions.

D3DLIGHT_SPOT
: Light is a spotlight source. This light is like a point light, except that the illumination is limited to a cone. This light type has a direction and several other parameters that determine the shape of the cone it produces. For information about these parameters, see the **D3DLIGHT8** structure.

D3DLIGHT_DIRECTIONAL
: Light is a directional source. This is equivalent to using a point light source at an infinite distance.

D3DLIGHT_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Directional lights are slightly faster than point light sources, but point lights look a little better. Spotlights offer interesting visual effects but are computationally expensive.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DLIGHT8

D3DMATERIALCOLORSOURCE

Defines the location at which a color or color component must be accessed for lighting calculations.

```
typedef enum _D3DMATERIALCOLORSOURCE {
    D3DMCS_MATERIAL         = 0,
    D3DMCS_COLOR1           = 1,
    D3DMCS_COLOR2           = 2,

    D3DMCS_FORCE_DWORD      = 0x7fffffff
} D3DMATERIALCOLORSOURCE;
```

Constants

D3DMCS_MATERIAL
Use the color from the current material.

D3DMCS_COLOR1
Use the diffuse vertex color.

D3DMCS_COLOR2
Use the specular vertex color.

D3DMCS_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

These flags are used to set the value of the following render states in the **D3DRENDERSTATETYPE** enumerated type.

- D3DRS_DIFFUSEMATERIALSOURCE
- D3DRS_SPECULARMATERIALSOURCE
- D3DRS_AMBIENTMATERIALSOURCE
- D3DRS_EMISSIVEMATERIALSOURCE

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE

D3DMULTISAMPLE_TYPE

Defines levels of full-scene multisampling that the device can apply.

```
typedef enum _D3DMULTISAMPLE_TYPE {
    D3DMULTISAMPLE_NONE          =  0,
    D3DMULTISAMPLE_2_SAMPLES     =  2,
    D3DMULTISAMPLE_3_SAMPLES     =  3,
    D3DMULTISAMPLE_4_SAMPLES     =  4,
    D3DMULTISAMPLE_5_SAMPLES     =  5,
    D3DMULTISAMPLE_6_SAMPLES     =  6,
    D3DMULTISAMPLE_7_SAMPLES     =  7,
    D3DMULTISAMPLE_8_SAMPLES     =  8,
    D3DMULTISAMPLE_9_SAMPLES     =  9,
    D3DMULTISAMPLE_10_SAMPLES    = 10,
    D3DMULTISAMPLE_11_SAMPLES    = 11,
    D3DMULTISAMPLE_12_SAMPLES    = 12,
    D3DMULTISAMPLE_13_SAMPLES    = 13,
    D3DMULTISAMPLE_14_SAMPLES    = 14,
    D3DMULTISAMPLE_15_SAMPLES    = 15,
    D3DMULTISAMPLE_16_SAMPLES    = 16,

    D3DMULTISAMPLE_FORCE_DWORD   = 0xffffffff
} D3DMULTISAMPLE_TYPE;
```

Constants

D3DMULTISAMPLE_NONE
: No level of full-scene multisampling is available.

D3DMULTISAMPLE_2_SAMPLES through D3DMULTISAMPLE_16_SAMPLES
: The level of full-scene multisampling available.

D3DMULTISAMPLE_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

In addition to enabling full-scene multisampling at **IDirect3DDevice8::Reset** time, there will be render states that turn various aspects on and off at fine-grained levels.

Multisampling is valid only on a swap chain that is being created or reset with the D3DSWAPEFFECT_DISCARD swap effect.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3D8::CheckDeviceMultiSampleType,
IDirect3DDevice8::CreateDepthStencilSurface,
IDirect3DDevice8::CreateRenderTarget, **D3DPRESENT_PARAMETERS**,
D3DSURFACE_DESC

D3DORDERTYPE

Defines the order type of a high-order surface.

```
typedef enum _D3DORDERTYPE {
  D3DORDER_LINEAR      = 1,
  D3DORDER_CUBIC       = 3,
  D3DORDER_QUINTIC     = 5,

  D3DORDER_FORCE_DWORD = 0x7fffffff
} D3DORDERTYPE;
```

Constants

D3DORDER_LINEAR
Linear order type.

D3DORDER_CUBIC
Cubic order type.

D3DORDER_QUINTIC
Quintic order type.

D3DORDER_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRECTPATCH_INFO, **D3DTRIPATCH_INFO**

D3DPATCHEDGESTYLE

Defines whether the current tessellation mode is discrete or continuous.

```
typedef enum _D3DPATCHEDGESTYLE {
  D3DPATCHEDGE_DISCRETE    = 0,
  D3DPATCHEDGE_CONTINUOUS  = 1,
```

```
    D3DPATCHEDGE_FORCE_DWORD = 0x7fffffff,
} D3DPATCHEDGESTYLE;
```

Constants

D3DPATCHEDGE_DISCRETE
: Discrete edge style. In discrete mode, you can specify float tessellation but it will be truncated to integers.

D3DPATCHEDGE_CONTINUOUS
: Continuous edge style. In continuous mode, tessellation is specified as float values which can be smoothly varied to reduce "popping" artifacts.

D3DPATCHEDGE_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Note that continuous tessellation produces a completely different tessellation pattern from the discrete one for the same tessellation values (this is more apparent in wire-frame mode). Thus, 4.0 continuous is not the same as 4 discrete.

Requirements

Header: Declared in D3d8types.h.

D3DPOOL

Defines the memory class that holds a resource's buffers.

```
typedef enum _D3DPOOL {
    D3DPOOL_DEFAULT             = 0,
    D3DPOOL_MANAGED             = 1,
    D3DPOOL_SYSTEMMEM           = 2,

    D3DPOOL_FORCE_DWORD         = 0x7fffffff
} D3DPOOL;
```

Constants

D3DPOOL_DEFAULT
: Resources are placed in the memory pool most appropriate for the set of usages requested for the given resource. This is usually video memory, including both local video memory and AGP memory. The D3DPOOL_DEFAULT pool is separate from D3DPOOL_MANAGED and D3DPOOL_SYTEMMEM, and it specifies that the resource is placed in the preferred memory for device access. Note that D3DPOOL_DEFAULT never indicates that either

D3DPOOL_MANAGED or D3DPOOL_SYSTEMMEM should be chosen as the memory pool type for this resource. Textures placed in the D3DPOOL_DEFAULT pool cannot be locked and are therefore not directly accessible. Instead, you must use functions such as **IDirect3Ddevice8::CopyRects** and **IDirect3dDevice8::UpdateTexture**. D3DPOOL_MANAGED is probably a better choice than D3DPOOL_DEFAULT for most applications. Note that some textures created in driver proprietary pixel formats, unknown to the Direct3D runtime, can be locked. Also note that—unlike textures—swapchain back-buffers, render targets, vertex buffers, and index buffers can be locked.

When a device is lost, resources created using D3DPOOL_DEFAULT must be released before calling **IDirect3DDevice8::Reset**. See Lost Devices for more information.

D3DPOOL_MANAGED
: Resources are copied automatically to device-accessible memory as needed. Managed resources are backed by system memory and do not need to be recreated when a device is lost. See Managing Resources for more information. Managed resources can be locked. Only the system-memory copy is directly modified. Direct3D copies your changes to driver-accessible memory as needed.

D3DPOOL_SYSTEMMEM
: Memory that is not typically accessible by the 3-D device. Consumes system RAM but does not reduce pageable RAM. These resources do not need to be recreated when a device is lost. Resources in this pool can be locked and can be used as the source for a **IDirect3DDevice8::CopyRects** or **IDirect3DDevice8::UpdateTexture** operation to a memory resource created with D3DPOOL_DEFAULT.

D3DPOOL_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Pools cannot be mixed for different objects contained within one resource (mip levels in a mipmap), and once a pool is chosen it cannot be changed.

Applications should use D3DPOOL_MANAGED for most static resources because this saves the application from having to deal with lost devices. (Managed resources are restored by the runtime.) This is especially beneficial for UMA systems. Dynamic texture resources can also be a good match for D3DPOOL_MANAGED, in spite of the high frequency at which they change. Other dynamic resources are not a good match for D3DPOOL_MANAGED. In fact, index buffers and vertex buffers cannot be created using D3DPOOL_MANAGED together with D3DUSAGE_DYNAMIC.

For dynamic textures, it is sometimes desirable to use a pair of video memory and system memory textures, allocating the video memory using D3DPOOL_DEFAULT and the system memory using D3DPOOL_SYSTEMMEM. You can lock and modify the bits of the system memory texture using a locking method. Then you can update the video memory texture using **IDirect3DDevice8::UpdateTexture**.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateCubeTexture, **IDirect3DDevice8::CreateIndexBuffer**, **IDirect3DDevice8::CreateTexture**, **IDirect3DDevice8::CreateVolumeTexture**, **IDirect3DDevice8::CreateVertexBuffer**, **D3DINDEXBUFFER_DESC**, **D3DSURFACE_DESC**, **D3DVERTEXBUFFER_DESC**, **D3DVOLUME_DESC**

D3DPRIMITIVETYPE

Defines the primitives supported by Microsoft® Direct3D®.

```
typedef enum _D3DPRIMITIVETYPE {
    D3DPT_POINTLIST           = 1,
    D3DPT_LINELIST            = 2,
    D3DPT_LINESTRIP           = 3,
    D3DPT_TRIANGLELIST        = 4,
    D3DPT_TRIANGLESTRIP       = 5,
    D3DPT_TRIANGLEFAN         = 6,

    D3DPT_FORCE_DWORD         = 0x7fffffff
} D3DPRIMITIVETYPE;
```

Constants

D3DPT_POINTLIST
: Renders the vertices as a collection of isolated points.

D3DPT_LINELIST
: Renders the vertices as a list of isolated straight line segments. Calls using this primitive type fail if the count is less than 2 or is odd.

D3DPT_LINESTRIP
: Renders the vertices as a single polyline. Calls using this primitive type fail if the count is less than 2.

D3DPT_TRIANGLELIST
: Renders the specified vertices as a sequence of isolated triangles. Each group of three vertices defines a separate triangle.

  Backface culling is affected by the current winding-order render state.

D3DPT_TRIANGLESTRIP
: Renders the vertices as a triangle strip. The backface-culling flag is automatically flipped on even-numbered triangles.

D3DPT_TRIANGLEFAN
: Renders the vertices as a triangle fan.

D3DPT_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Using Triangle Strips or Triangle Fans is often more efficient than using triangle lists because fewer vertices are duplicated.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::DrawIndexedPrimitive,
IDirect3DDevice8::DrawIndexedPrimitiveUP,
IDirect3DDevice8::DrawPrimitive, **IDirect3DDevice8::DrawPrimitiveUP**

D3DRENDERSTATETYPE

Defines device render states

```
typedef enum _D3DRENDERSTATETYPE {
    D3DRS_ZENABLE                =   7,
    D3DRS_FILLMODE               =   8,
    D3DRS_SHADEMODE              =   9,
    D3DRS_LINEPATTERN            =  10,
    D3DRS_ZWRITEENABLE           =  14,
    D3DRS_ALPHATESTENABLE        =  15,
    D3DRS_LASTPIXEL              =  16,
    D3DRS_SRCBLEND               =  19,
    D3DRS_DESTBLEND              =  20,
    D3DRS_CULLMODE               =  22,
    D3DRS_ZFUNC                  =  23,
    D3DRS_ALPHAREF               =  24,
    D3DRS_ALPHAFUNC              =  25,
    D3DRS_DITHERENABLE           =  26,
    D3DRS_ALPHABLENDENABLE       =  27,
    D3DRS_FOGENABLE              =  28,
    D3DRS_SPECULARENABLE         =  29,
    D3DRS_ZVISIBLE               =  30,
    D3DRS_FOGCOLOR               =  34,
    D3DRS_FOGTABLEMODE           =  35,
    D3DRS_FOGSTART               =  36,
    D3DRS_FOGEND                 =  37,
    D3DRS_FOGDENSITY             =  38,
```

```
D3DRS_EDGEANTIALIAS          =  40,
D3DRS_ZBIAS                  =  47,
D3DRS_RANGEFOGENABLE         =  48,

D3DRS_STENCILENABLE          =  52,
D3DRS_STENCILFAIL            =  53,
D3DRS_STENCILZFAIL           =  54,
D3DRS_STENCILPASS            =  55,
D3DRS_STENCILFUNC            =  56,
D3DRS_STENCILREF             =  57,
D3DRS_STENCILMASK            =  58,
D3DRS_STENCILWRITEMASK       =  59,
D3DRS_TEXTUREFACTOR          =  60,

D3DRS_WRAP0                  = 128,
D3DRS_WRAP1                  = 129,
D3DRS_WRAP2                  = 130,
D3DRS_WRAP3                  = 131,
D3DRS_WRAP4                  = 132,
D3DRS_WRAP5                  = 133,
D3DRS_WRAP6                  = 134,
D3DRS_WRAP7                  = 135,
D3DRS_CLIPPING               = 136,
D3DRS_LIGHTING               = 137,
D3DRS_AMBIENT                = 139,
D3DRS_FOGVERTEXMODE          = 140,
D3DRS_COLORVERTEX            = 141,
D3DRS_LOCALVIEWER            = 142,
D3DRS_NORMALIZENORMALS       = 143,
D3DRS_DIFFUSEMATERIALSOURCE  = 145,
D3DRS_SPECULARMATERIALSOURCE = 146,
D3DRS_AMBIENTMATERIALSOURCE  = 147,
D3DRS_EMISSIVEMATERIALSOURCE = 148,
D3DRS_VERTEXBLEND            = 151,
D3DRS_CLIPPLANEENABLE        = 152,

D3DRS_SOFTWAREVERTEXPROCESSING = 153,

D3DRS_POINTSIZE              = 154,
D3DRS_POINTSIZE_MIN          = 155,
D3DRS_POINTSPRITEENABLE      = 156,

D3DRS_POINTSCALEENABLE       = 157,
D3DRS_POINTSCALE_A           = 158,
D3DRS_POINTSCALE_B           = 159,
D3DRS_POINTSCALE_C           = 160,
```

```
    D3DRS_MULTISAMPLEANTIALIAS     = 161,
    D3DRS_MULTISAMPLEMASK          = 162,

    D3DRS_PATCHEDGESTYLE           = 163,
    D3DRS_PATCHSEGMENTS            = 164,

    D3DRS_DEBUGMONITORTOKEN        = 165,
    D3DRS_POINTSIZE_MAX            = 166,
    D3DRS_INDEXEDVERTEXBLENDENABLE  = 167,
    D3DRS_COLORWRITEENABLE         = 168,
    D3DRS_TWEENFACTOR              = 170,
    D3DRS_BLENDOP                  = 171,

    D3DRS_FORCE_DWORD              = 0x7fffffff
} D3DRENDERSTATETYPE;
```

Constants

D3DRS_ZENABLE

Depth-buffering state as one member of the **D3DZBUFFERTYPE** enumerated type. Set this state to D3DZB_TRUE to enable z-buffering, D3DZB_USEW to enable w-buffering, or D3DZB_FALSE to disable depth buffering.

The default value for this render state is D3DZB_TRUE if a depth stencil was created along with the swap chain by setting the **EnableAutoDepthStencil** member of the **D3DPRESENT_PARAMETERS** structure to TRUE, and D3DZB_FALSE otherwise.

D3DRS_FILLMODE

One or more members of the **D3DFILLMODE** enumerated type. The default value is D3DFILL_SOLID.

D3DRS_SHADEMODE

One or more members of the **D3DSHADEMODE** enumerated type. The default value is D3DSHADE_GOURAUD.

D3DRS_LINEPATTERN

D3DLINEPATTERN structure. The default values are 0 for **wRepeatPattern**, and 0 for **wLinePattern**.

D3DRS_ZWRITEENABLE

TRUE to enable the application to write to the depth buffer. The default value is TRUE. This member enables an application to prevent the system from updating the depth buffer with new depth values. If FALSE, depth comparisons are still made according to the render state D3DRS_ZFUNC, assuming that depth buffering is taking place, but depth values are not written to the buffer.

D3DRS_ALPHATESTENABLE

TRUE to enable alpha tests. The default value is FALSE. This member enables applications to turn off the tests that accept or reject a pixel, based on its alpha value.

The incoming alpha value is compared with the reference alpha value, using the comparison function provided by the D3DRS_ALPHAFUNC render state. When this mode is enabled, alpha blending occurs only if the test succeeds.

D3DRS_LASTPIXEL
FALSE to enable drawing the last pixel in a line or triangle. The default value is TRUE.

D3DRS_SRCBLEND
One member of the **D3DBLEND** enumerated type. The default value is D3DBLEND_ONE.

D3DRS_DESTBLEND
One member of the **D3DBLEND** enumerated type. The default value is D3DBLEND_ZERO.

D3DRS_CULLMODE
Specifies how back-facing triangles are culled, if at all. This can be set to one member of the **D3DCULL** enumerated type. The default value is D3DCULL_CCW.

D3DRS_ZFUNC
One member of the **D3DCMPFUNC** enumerated type. The default value is D3DCMP_LESSEQUAL. This member enables an application to accept or reject a pixel, based on its distance from the camera.

The depth value of the pixel is compared with the depth-buffer value. If the depth value of the pixel passes the comparison function, the pixel is written.

The depth value is written to the depth buffer only if the render state is TRUE.

Software rasterizers and many hardware accelerators work faster if the depth test fails, because there is no need to filter and modulate the texture if the pixel is not going to be rendered.

D3DRS_ALPHAREF
Value that specifies a reference alpha value against which pixels are tested when alpha-testing is enabled. This is an 8-bit value placed in the low 8 bits of the **DWORD** render-state value. Values can range from 0x00000000 through 0x000000FF.

D3DRS_ALPHAFUNC
One member of the **D3DCMPFUNC** enumerated type. The default value is D3DCMP_ALWAYS. This member enables an application to accept or reject a pixel, based on its alpha value.

D3DRS_DITHERENABLE
TRUE to enable dithering. The default value is FALSE.

D3DRS_ALPHABLENDENABLE
TRUE to enable alpha-blended transparency. The default value is FALSE.

The type of alpha blending is determined by the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states.

Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **DevCaps** member of the **D3DCAPS8** structure to find out whether this render state is supported.

D3DRS_FOGENABLE
TRUE to enable fog blending. The default value is FALSE. For more information on using fog blending, see Fog.

D3DRS_SPECULARENABLE
TRUE to enable specular highlights. The default value is FALSE.

Specular highlights are calculated as though every vertex in the object being lit were at the object's origin. This gives the expected results as long as the object is modeled around the origin and the distance from the light to the object is relatively large. In other cases the results as undefined.

When this member is set to TRUE, the specular color is added to the base color after the texture cascade but before alpha blending.

D3DRS_ZVISIBLE
Not supported.

D3DRS_FOGCOLOR
Value whose type is **D3DCOLOR**. The default value is 0. For more information on fog color, see Fog Color.

D3DRS_FOGTABLEMODE
The fog formula to be used for pixel fog. Set to one of the members of the **D3DFOGMODE** enumerated type. The default value is D3DFOG_NONE. For more information on pixel fog, see Pixel Fog.

D3DRS_FOGSTART
Depth at which pixel or vertex fog effects begin for linear fog mode. Depth is specified in world space for vertex fog, and either device space [0.0, 1.0] or world space for pixel fog. For pixel fog, these values are in device space when the system uses z for fog calculations, and world-space when the system is using eye-relative fog (w-fog). For more information, see Fog Parameters and Eye-Relative vs. Z-based Depth.

Values for this render state are floating-point values. Because the **IDirect3DDevice8::SetRenderState** method accepts **DWORD** values, your application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_FOGSTART, *((DWORD*) (&fFogStart)));
```

D3DRS_FOGEND
Depth at which pixel or vertex fog effects end for linear fog mode. Depth is specified in world space for vertex fog, and either device space [0.0, 1.0] or world space for pixel fog. For pixel fog, these values are in device space when the system uses z for fog calculations, and world-space when the system is using eye-relative fog (w-fog). For more information, see Fog Parameters and Eye-Relative vs. Z-based Depth.

Values for this render state are floating-point values. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, your

application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_FOGEND, *((DWORD*) (&fFogEnd)));
```

D3DRS_FOGDENSITY

Fog density for pixel or vertex fog used in the exponential fog modes (D3DFOG_EXP and D3DFOG_EXP2). Valid density values range from 0.0 through 1.0. The default value is 1.0. For more information, see Fog Parameters.

Values for this render state are floating-point values. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, your application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_FOGDENSITY, *((DWORD*) (&fFogDensity)));
```

D3DRS_EDGEANTIALIAS

TRUE to antialias lines forming the convex outline of objects. The default value is FALSE. If TRUE, applications should render only lines, and only to the exterior edges of polygons in a scene. The behavior is undefined if triangles or points are drawn when this render state is set. Antialiasing is performed by averaging the values of neighboring pixels. Although this is not the best way to perform antialiasing, it can be very efficient; hardware that supports this kind of operation is becoming more common.

You can enable edge antialiasing only on devices that expose the D3DPRASTERCAPS_ANTIALIASEDGES capability. For more information on edge antialiasing, see Edge Antialiasing and Antialiasing State.

D3DRS_ZBIAS

Integer value in the range 0 through 16 that causes polygons that are physically coplanar to appear separate. Polygons with a high z-bias value appear in front of polygons with a low value, without requiring sorting for drawing order. Polygons with a value of 1 appear in front of polygons with a value of 0, and so on. The default value is 0. For more information, see Using Depth Buffers.

D3DRS_RANGEFOGENABLE

TRUE to enable range-based vertex fog. The default value is FALSE, in which case the system uses depth-based fog. In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene. In range-based fog, all fog methods work as usual, except that they use range instead of depth in the computations.

Range is the correct factor to use for fog computations, but depth is commonly used instead because range is expensive to compute and depth is generally already available. Using depth to calculate fog has the undesirable effect of having the fogginess of peripheral objects change as the viewer's eye moves—in this case, the depth changes, and the range remains constant.

Because no hardware currently supports per-pixel range-based fog, range correction is offered only for vertex fog.

For more information, see Range-Based Fog and Vertex Fog.

D3DRS_STENCILENABLE
TRUE to enable stenciling, or FALSE to disable stenciling. The default value is FALSE.

For more information, see Stencil Buffers.

D3DRS_STENCILFAIL
Stencil operation to perform if the stencil test fails. This can be one member of the **D3DSTENCILOP** enumerated type. The default value is D3DSTENCILOP_KEEP.

For more information, see Stencil Buffers.

D3DRS_STENCILZFAIL
Stencil operation to perform if the stencil test passes and the depth test (z-test) fails. This can be one of the members of the **D3DSTENCILOP** enumerated type. The default value is D3DSTENCILOP_KEEP.

D3DRS_STENCILPASS
Stencil operation to perform if both the stencil and the depth (z) tests pass. This can be one member of the **D3DSTENCILOP** enumerated type. The default value is D3DSTENCILOP_KEEP.

D3DRS_STENCILFUNC
Comparison function for the stencil test. This can be one member of the **D3DCMPFUNC** enumerated type. The default value is D3DCMP_ALWAYS.

The comparison function is used to compare the reference value to a stencil buffer entry. This comparison applies only to the bits in the reference value and stencil buffer entry that are set in the stencil mask (set by the D3DRS_STENCILMASK render state). If TRUE, the stencil test passes.

D3DRS_STENCILREF
Integer reference value for the stencil test. The default value is 0.

D3DRS_STENCILMASK
Mask applied to the reference value and each stencil buffer entry to determine the significant bits for the stencil test. The default mask is 0xFFFFFFFF.

D3DRS_STENCILWRITEMASK
Write mask applied to values written into the stencil buffer. The default mask is 0xFFFFFFFF.

D3DRS_TEXTUREFACTOR
Color used for multiple-texture blending with the D3DTA_TFACTOR texture-blending argument or the D3DTOP_BLENDFACTORALPHA texture-blending operation. The associated value is a **D3DCOLOR** variable.

D3DRS_WRAP0 through D3DRS_WRAP7
Texture-wrapping behavior for multiple sets of texture coordinates. Valid values for these render states can be any combination of the D3DWRAPCOORD_0 (or D3DWRAP_U), D3DWRAPCOORD_1 (or D3DWRAP_V), D3DWRAPCOORD_2 (or D3DWRAP_W), and D3DWRAPCOORD_3 flags. These cause the system to wrap in the direction of the first, second, third, and fourth dimensions, sometimes referred to at the s, t, r, and q directions, for a

given texture. The default value for these render states is 0 (wrapping disabled in all directions).

D3DRS_CLIPPING
TRUE to enable primitive clipping by Microsoft® Direct3D®, or FALSE to disable it. The default value is TRUE.

D3DRS_LIGHTING
TRUE to enable Direct3D lighting, or FALSE to disable it. The default value is TRUE. Only vertices that include a vertex normal are properly lit; vertices that do not contain a normal employ a dot product of 0 in all lighting calculations.

D3DRS_AMBIENT
Ambient light color. This value is of type **D3DCOLOR**. The default value is 0.

D3DRS_FOGVERTEXMODE
Fog formula to be used for vertex fog. Set to one member of the **D3DFOGMODE** enumerated type. The default value is D3DFOG_NONE.

D3DRS_COLORVERTEX
TRUE to enable per-vertex color, or FALSE to disable it. The default value is TRUE. Enabling per-vertex color allows the system to include the color defined for individual vertices in its lighting calculations.

For more information, see the following render states.

- D3DRS_DIFFUSEMATERIALSOURCE
- D3DRS_SPECULARMATERIALSOURCE
- D3DRS_AMBIENTMATERIALSOURCE
- D3DRS_EMISSIVEMATERIALSOURCE

D3DRS_LOCALVIEWER
TRUE to enable camera-relative specular highlights, or FALSE to use orthogonal specular highlights. The default value is TRUE. Applications that use orthogonal projection should specify false.

D3DRS_NORMALIZENORMALS
TRUE to enable automatic normalization of vertex normals, or FALSE to disable it. The default value is FALSE. Enabling this feature causes the system to normalize the vertex normals for vertices after transforming them to camera space, which can be computationally expensive.

D3DRS_DIFFUSEMATERIALSOURCE
Diffuse color source for lighting calculations. Valid values are members of the **D3DMATERIALCOLORSOURCE** enumerated type. The default value is D3DMCS_COLOR1. The value for this render state is used only if the D3DRS_COLORVERTEX render state is set to TRUE.

D3DRS_SPECULARMATERIALSOURCE
Specular color source for lighting calculations. Valid values are members of the **D3DMATERIALCOLORSOURCE** enumerated type. The default value is D3DMCS_COLOR2.

D3DRS_AMBIENTMATERIALSOURCE

Ambient color source for lighting calculations. Valid values are members of the **D3DMATERIALCOLORSOURCE** enumerated type. The default value is D3DMCS_COLOR2.

D3DRS_EMISSIVEMATERIALSOURCE

Emissive color source for lighting calculations. Valid values are members of the **D3DMATERIALCOLORSOURCE** enumerated type. The default value is D3DMCS_MATERIAL.

D3DRS_VERTEXBLEND

Number of matrices to use to perform geometry blending, if any. Valid values are members of the **D3DVERTEXBLENDFLAGS** enumerated type. The default value is D3DVBF_DISABLE.

D3DRS_CLIPPLANEENABLE

Enables or disables user-defined clipping planes. Valid values are any **DWORD** in which the status of each bit (set or not set) toggles the activation state of a corresponding user-defined clipping plane. The least significant bit (bit 0) controls the first clipping plane at index 0, and subsequent bits control the activation of clipping planes at higher indexes. If a bit is set, the system applies the appropriate clipping plane during scene rendering. The default value is 0.

The **D3DCLIPPLANE*n*** macros are defined to provide a convenient way to enable clipping planes.

D3DRS_SOFTWAREVERTEXPROCESSING

BOOL value that enables applications to query and select hardware or software vertex processing. For a D3DDEVTYPE_SW device type this value is fixed to TRUE. This value can be set by the application for a D3DDEVTYPE_REF device type. For a D3DDEVTYPE_HAL device type, this value can be set only by the application when D3DDEVCAPS_HWTRANSFORMANDLIGHT is set; otherwise this flag is fixed to TRUE. When variable, the default value is FALSE.

Changing the vertex processing render state in mixed vertex processing mode will reset the current stream, indices, and vertex shader to their default values of NULL or 0.

D3DRS_POINTSIZE

Float value that specifies the size to use for point size computation in cases where point size is not specified for each vertex. This value is not used when the vertex contains point size. This value is in screen space units if D3DRS_POINTSCALEENABLE is FALSE; otherwise this value is in world space units. The default value is 1.0f. The range for this value is greater than or equal to 0.0f. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, you application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_PATCHSEGMENTS, *((DWORD*)&PointSize));
```

D3DRS_POINTSIZE_MIN

Float value that specifies the minimum size of point primitives. Point primitives are clamped to this size during rendering. Setting this to values smaller than 1.0 results in points dropping out when the point does not cover a pixel center and

antialiasing is disabled or being rendered with reduced intensity when antialiasing is enabled. The default value is 1.0f. The range for this value is greater than or equal to 0.0f. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, you application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_PATCHSEGMENTS,
*((DWORD*)&PointSizeMin));
```

D3DRS_POINTSPRITEENABLE

BOOL value. When TRUE, texture coordinates of point primitives are set so that full textures are mapped on each point. When FALSE, the vertex texture coordinates are used for the entire point. The default value is FALSE. You can achieve DirectX 7 style single-pixel points by setting D3DRS_POINTSCALEENABLE to FALSE and D3DRS_POINTSIZE to 1.0, which are the default values.

D3DRS_POINTSCALEENABLE

BOOL value that controls computation of size for point primitives. When TRUE, the point size is interpreted as a camera space value and is scaled by the distance function and the frustum to viewport Y axis scaling to compute the final screen space point size. When FALSE, the point size is interpreted as screen space and used directly. The default value is FALSE.

D3DRS_POINTSCALE_A

Float value that controls for distance-based size attenuation for point primitives. Active only whenD3DRS_POINTSCALEENABLE is TRUE. The default value is 1.0f. The range for this value is greater than or equal to 0.0f. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, you application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_PATCHSEGMENTS,
*((DWORD*)&PointScaleA));
```

D3DRS_POINTSCALE_B

Float value that controls for distance-based size attenuation for point primitives. Active only when D3DRS_POINTSCALEENABLE is TRUE. The default value is 0.0f. The range for this value is greater than or equal to 0.0f. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, you application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_PATCHSEGMENTS,
*((DWORD*)&PointScaleB));
```

D3DRS_POINTSCALE_C

Float value that controls for distance-based size attenuation for point primitives. Active only when D3DRS_POINTSCALEENABLE is TRUE. The default value is 0.0f. The range for this value is greater than or equal to 0.0f. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, you

application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_PATCHSEGMENTS,
*((DWORD*)&PointScaleC));
```

D3DRS_MULTISAMPLEANTIALIAS

BOOL value that determines how individual samples are computed when using a multisample render target buffer. When set to TRUE, the multiple samples are computed so that full-scene antialiasing is performed by sampling at different sample positions for each multiple sample. When set to FALSE, the multiple samples are all written with the same sample value—sampled at the pixel center, which allows non-antialiased rendering to a multisample buffer. This render state has no effect when rendering to a single sample buffer. The default value is TRUE.

D3DRS_MULTISAMPLEMASK

Each bit in this mask, starting at the LSB, controls modification of one of the samples in a multisample render target. Thus, for an 8-sample render target, the low byte contains the 8 write enables for each of the 8 samples. This render state has no effect when rendering to a single sample buffer. The default value is 0xFFFFFFFF.

This render state enables use of a multisample buffer as an accumulation buffer, doing multipass rendering of geometry where each pass updates a subset of samples.

D3DRS_PATCHEDGESTYLE

Sets whether patch edges will use float style tessellation. Possible values are defined by the **D3DPATCHEDGESTYLE** enumerated type.

D3DRS_PATCHSEGMENTS

Sets the number of segments as a float value for each edge when drawing patches. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, your application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_PATCHSEGMENTS,
*((DWORD*)&NumSegments));
```

D3DRS_DEBUGMONITORTOKEN

Set only for debugging the monitor. Possible values are defined by the **D3DDEBUGMONITORTOKENS** enumerated type. Note that if D3DRS_DEBUGMONITORTOKEN is set, the call is treated as passing a token to the debug monitor. For example, if, after passing D3DDMT_ENABLE or D3DDMT_DISABLE to D3DRS_DEBUGMONITORTOKEN, other token values are passed in, the state (enabled or disabled) of the debug monitor will still persist.

This state is only useful for debug builds. The debug monitor defaults to D3DDMT_ENABLE.

D3DRS_POINTSIZE_MAX

Float value that specifies the maximum size to which point sprites will be clamped. The value must be less than or equal to the **MaxPointSize** member of **D3DCAPS8** and greater than or equal to D3DRS_POINTSIZE_MIN. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, you application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_PONTSIZE_MAX, *((DWORD*)&PointSizeMax));
```

D3DRS_INDEXEDVERTEXBLENDENABLE

BOOL value that enables or disables indexed vertex blending. When set to TRUE, indexed vertex blending is enabled. When set to FALSE, indexed vertex blending is disabled. If this render state is enabled, the user must pass matrix indices as a packed **DWORD** with every vertex. When the render state is disabled and vertex blending is enabled through the D3DRS_VERTEXBLEND state, it is equivalent to having matrix indices 0, 1, 2, 3 in every vertex.

D3DRS_COLORWRITEENABLE

UINT value that enables a per-channel write for the render target color buffer. A set bit results in the color channel being updated during 3-D rendering. A clear bit results in the color channel being unaffected. This functionality is available if the D3DPMISCCAPS_COLORWRITEENABLE capabilities bit is set in the **PrimitiveMiscCaps** member of the **D3DCAPS8** structure for the device. This render state does not affect the clear operation. The default value is 0x0000000F.

Valid values for this render state can be any combination of the D3DCOLORWRITEENABLE_ALPHA, D3DCOLORWRITEENABLE_BLUE, D3DCOLORWRITEENABLE_GREEN, or D3DCOLORWRITEENABLE_RED flags.

D3DRS_TWEENFACTOR

Float value that controls the tween factor. Because the **IDirect3DDevice8::SetRenderState** method accepts DWORD values, you application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice8->SetRenderState(D3DRS_PATCHSEGMENTS,
*((DWORD*)&TweenFactor));
```

D3DRS_BLENDOP

Value used to select the arithmetic operation applied when the alpha blending render state, D3DRS_ALPHABLENDENABLE, is set to TRUE. Valid values are defined by the **D3DBLENDOP** enumerated type. The default value is D3DBLENDOP_ADD.

If the D3DPMISCCAPS_BLENDOP device capability is not supported, then D3DBLENDOP_ADD is performed.

D3DRS_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Direct3D defines the D3DRENDERSTATE_WRAPBIAS constant as a convenience for applications to enable or disable texture wrapping, based on the zero-based integer of a texture coordinate set (rather than explicitly using one of the D3DRS_WRAP*n* state values). Add the D3DRENDERSTATE_WRAPBIAS value to the zero-based index of a texture coordinate set to calculate the D3DRS_WRAP*n* value that corresponds to that index, as shown in the following example.

```
// Enable U/V wrapping for textures that use the texture
// coordinate set at the index within the dwIndex variable.
HRESULT hr = pd3dDevice->SetRenderState(
    dwIndex + D3DRENDERSTATE_WRAPBIAS,
    D3DWRAPCOORD_0 | D3DWRAPCOORD_1);

// If dwIndex is 3, the value that results from
// the addition equals D3DRS_WRAP3 (131).
```

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetRenderState, **IDirect3DDevice8::SetRenderState**

D3DRESOURCETYPE

Defines resource types.

```
typedef enum _D3DRESOURCETYPE {
    D3DRTYPE_SURFACE             = 1,
    D3DRTYPE_VOLUME              = 2,
    D3DRTYPE_TEXTURE             = 3,
    D3DRTYPE_VOLUMETEXTURE       = 4,
    D3DRTYPE_CUBETEXTURE         = 5,
    D3DRTYPE_VERTEXBUFFER        = 6,
    D3DRTYPE_INDEXBUFFER         = 7,

    D3DRTYPE_FORCE_DWORD         = 0x7fffffff
} D3DRESOURCETYPE;
```

Constants

D3DRTYPE_SURFACE
Surface resource.

D3DRTYPE_VOLUME
Volume resource.

D3DRTYPE_TEXTURE
Texture resource.

D3DRTYPE_VOLUMETEXTURE
Volume texture resource.

D3DRTYPE_CUBETEXTURE
Cube texture resource.

D3DRTYPE_VERTEXBUFFER
Vertex buffer resource.

D3DRTYPE_INDEXBUFFER
Index buffer resource.

D3DRTYPE_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DResource8::GetType

D3DSHADEMODE

Defines constants that describe the supported shading modes.

```
typedef enum _D3DSHADEMODE {
    D3DSHADE_FLAT           = 1,
    D3DSHADE_GOURAUD        = 2,
    D3DSHADE_PHONG          = 3,

    D3DSHADE_FORCE_DWORD    = 0x7fffffff
} D3DSHADEMODE;
```

Constants

D3DSHADE_FLAT
Flat shading mode. The color and specular component of the first vertex in the triangle are used to determine the color and specular component of the face. These colors remain constant across the triangle; that is, they are not interpolated. The specular alpha is interpolated. See Remarks.

D3DSHADE_GOURAUD
Gouraud shading mode. The color and specular components of the face are determined by a linear interpolation between all three of the triangle's vertices.

D3DSHADE_PHONG
Not supported.

D3DSHADE_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The first vertex of a triangle for flat shading mode is defined in the following manner.

- For a triangle list, the first vertex of the triangle *i* is $i * 3$.
- For a triangle strip, the first vertex of the triangle *i* is vertex *i*.
- For a triangle fan, the first vertex of the triangle *i* is vertex $i + 1$.

The members of this enumerated type define the vales for the D3DRS_SHADEMODE render state.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE

D3DSTATEBLOCKTYPE

Defines logical groups of device states.

```
typedef enum _D3DSTATEBLOCKTYPE {
    D3DSBT_ALL          = 1,
    D3DSBT_PIXELSTATE   = 2,
    D3DSBT_VERTEXSTATE  = 3,

    D3DSBT_FORCE_DWORD  = 0xffffffff
} D3DSTATEBLOCKTYPE;
```

Constants

D3DSBT_ALL
Capture all of the following device states.

All current render states.	All current clipplanes.
All current texture stage states.	The current material.
All current textures.	All current lights and enabled light parameters.
The current palette.	The current pixel shader.

All current streams.
The current pixel shader constants.
The current viewport.
The current vertex shader.
All current transforms.
The current vertex shader constants.

D3DSBT_PIXELSTATE
Capture the current pixel shader and pixel shader constants, as well as all of the following pixel-related device states.

Render States

D3DRS_ALPHABLENDENABLE
D3DRS_ALPHAFUNC
D3DRS_ALPHAREF
D3DRS_ALPHATESTENABLE
D3DRS_BLENDOP
D3DRS_COLORWRITEENABLE
D3DRS_DESTBLEND
D3DRS_DITHERENABLE
D3DRS_EDGEANTIALIAS
D3DRS_FILLMODE
D3DRS_FOGDENSITY
D3DRS_FOGEND
D3DRS_FOGSTART
D3DRS_LASTPIXEL
D3DRS_LINEPATTERN
D3DRS_SHADEMODE
D3DRS_SRCBLEND
D3DRS_STENCILENABLE
D3DRS_STENCILFAIL
D3DRS_STENCILFUNC
D3DRS_STENCILMASK
D3DRS_STENCILPASS
D3DRS_STENCILREF
D3DRS_STENCILWRITEMASK
D3DRS_STENCILZFAIL
D3DRS_TEXTUREFACTOR
D3DRS_WRAP0 through D3DRS_WRAP7
D3DRS_ZBIAS
D3DRS_ZENABLE
D3DRS_ZFUNC
D3DRS_ZWRITEENABLE

Texture Stage States

D3DTSS_ADDRESSU
D3DTSS_ADDRESSV
D3DTSS_ADDRESSW
D3DTSS_ALPHAARG0
D3DTSS_ALPHAARG1
D3DTSS_ALPHAARG2
D3DTSS_ALPHAOP
D3DTSS_BORDERCOLOR
D3DTSS_BUMPENVLOFFSET
D3DTSS_BUMPENVLSCALE
D3DTSS_BUMPENVMAT00
D3DTSS_BUMPENVMAT01
D3DTSS_BUMPENVMAT10
D3DTSS_BUMPENVMAT11
D3DTSS_COLORARG0
D3DTSS_COLORARG1
D3DTSS_COLORARG2
D3DTSS_COLOROP
D3DTSS_MAGFILTER
D3DTSS_MAXANISOTROPY
D3DTSS_MAXMIPLEVEL
D3DTSS_MINFILTER
D3DTSS_MIPFILTER
D3DTSS_MIPMAPLODBIAS

D3DTSS_RESULTARG
D3DTSS_TEXCOORDINDEX
D3DTSS_TEXTURETRANSFORMFLAGS

D3DSBT_VERTEXSTATE
Capture all the current lights, the current vertex shader and vertex shader constants, and the texture stage states specified by D3DTSS_TEXCOORDINDEX and D3DTSS_TEXTURETRANSFORMFLAGS. In addition, this flag captures all of the following vertex-related device states.

Render States

D3DRS_AMBIENT
D3DRS_AMBIENTMATERIALSOURCE
D3DRS_CLIPPING
D3DRS_CLIPPLANEENABLE
D3DRS_COLORVERTEX
D3DRS_DIFFUSEMATERIALSOURCE
D3DRS_EMISSIVEMATERIALSOURCE
D3DRS_FOGDENSITY
D3DRS_FOGEND
D3DRS_FOGSTART
D3DRS_FOGTABLEMODE
D3DRS_FOGVERTEXMODE
D3DRS_INDEXEDVERTEXBLENDENABLE
D3DRS_LIGHTING
D3DRS_LOCALVIEWER
D3DRS_MULTISAMPLEANTIALIAS
D3DRS_MULTISAMPLEMASK
D3DRS_NORMALIZENORMALS
D3DRS_PATCHEDGESTYLE
D3DRS_PATCHSEGMENTS
D3DRS_POINTSCALE_A
D3DRS_POINTSCALE_B
D3DRS_POINTSCALE_C
D3DRS_POINTSCALEENABLE
D3DRS_POINTSIZE
D3DRS_POINTSIZE_MAX
D3DRS_POINTSIZE_MIN
D3DRS_POINTSPRITEENABLE
D3DRS_RANGEFOGENABLE
D3DRS_SOFTWAREVERTEXPROCESSING
D3DRS_SPECULARMATERIALSOURCE
D3DRS_TWEENFACTOR
D3DRS_VERTEXBLEND

D3DSBT_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The D3DSBT_PIXELSTATE and D3DSBT_VERTEXSTATE values identify different logical groups of device states, though some states are common to both

groups. The union of D3DSBT_PIXELSTATE and D3DSBT_VERTEXSTATE is not equal to D3DSBT_ALL. The D3DSBT_PIXELSTATE and D3DSBT_VERTEXSTATE values enable the capture of these frequently modified states between calls to **IDirect3Ddevice8::DrawPrimitive** without incurring the performance penalty of capturing the entire state.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::CreateStateBlock

D3DSTENCILOP

Defines stencil operations.

```
typedef enum _D3DSTENCILOP {
    D3DSTENCILOP_KEEP          = 1,
    D3DSTENCILOP_ZERO          = 2,
    D3DSTENCILOP_REPLACE       = 3,
    D3DSTENCILOP_INCRSAT       = 4,
    D3DSTENCILOP_DECRSAT       = 5,
    D3DSTENCILOP_INVERT        = 6,
    D3DSTENCILOP_INCR          = 7,
    D3DSTENCILOP_DECR          = 8,

    D3DSTENCILOP_FORCE_DWORD    = 0x7fffffff
} D3DSTENCILOP;
```

Constants

D3DSTENCILOP_KEEP
Do not update the entry in the stencil buffer. This is the default value.

D3DSTENCILOP_ZERO
Set the stencil-buffer entry to 0.

D3DSTENCILOP_REPLACE
Replace the stencil-buffer entry with reference value.

D3DSTENCILOP_INCRSAT
Increment the stencil-buffer entry, clamping to the maximum value. See Remarks for information on the maximum stencil-buffer values.

D3DSTENCILOP_DECRSAT
Decrement the stencil-buffer entry, clamping to zero.

D3DSTENCILOP_INVERT

Invert the bits in the stencil-buffer entry.

D3DSTENCILOP_INCR
Increment the stencil-buffer entry, wrapping to zero if the new value exceeds the maximum value. See Remarks for information on the maximum stencil-buffer values.

D3DSTENCILOP_DECR
Decrement the stencil-buffer entry, wrapping to the maximum value if the new value is less than zero.

D3DSTENCILOP_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Stencil-buffer entries are integer values ranging from 0 through $2^n - 1$, where n is the bit depth of the stencil buffer.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE

D3DSWAPEFFECT

Defines swap effects.

```
typedef enum _D3DSWAPEFFECT {
    D3DSWAPEFFECT_DISCARD        = 1,
    D3DSWAPEFFECT_FLIP           = 2,
    D3DSWAPEFFECT_COPY           = 3,
    D3DSWAPEFFECT_COPY_VSYNC     = 4,

    D3DSWAPEFFECT_FORCE_DWORD    = 0xFFFFFFFF
} D3DSWAPEFFECT;
```

Constants

D3DSWAPEFFECT_DISCARD
When a swap chain is created with a swap effect of D3DSWAPEFFECT_FLIP, D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC, the runtime will guarantee that a **IDirect3DDevice8::Present** operation will not affect the content of any of the back buffers. Unfortunately, meeting this guarantee can involve substantial video memory or processing overheads, especially when implementing flip semantics for a windowed swap chain or copy

semantics for a full-screen swap chain. An application may use the D3DSWAPEFFECT_DISCARD swap effect to avoid these overheads and to enable the display driver to select the most efficient presentation technique for the swap chain. This is also the only swap effect that may be used when specifying a value other than D3DMULTISAMPLE_NONE for the **MultiSampleType** member of **D3DPRESENT_PARAMETERS**.

Like a swap chain that uses D3DSWAPEFFECT_FLIP, a swap chain that uses D3DSWAPEFFECT_DISCARD might include more than one back buffer, any of which may be accessed using **IDirect3DDevice8::GetBackBuffer** or **IDirect3DSwapChain8::GetBackBuffer**. The swap chain is best envisaged as a queue in which 0 always indexes the back buffer that will be displayed by the next Present operation and from which buffers are discarded once they have been displayed.

An application that uses this swap effect cannot make any assumptions about the contents of a discarded back buffer and should therefore update an entire back buffer before invoking a **Present** operation that would display it. Although this is not enforced, the debug version of the runtime will overwrite the contents of discarded back buffers with random data to enable developers to verify that their applications are updating the entire back buffer surfaces correctly.

For a full-screen swap chain, the presentation rate is determined by the value assigned to the **FullScreen_PresentationInterval** member of the **D3DPRESENT_PARAMETERS** structure when the device or swap chain is created. Unless this value is D3DPRESENT_INTERVAL_IMMEDIATE, the presentation will be synchronized with the vertical sync of the monitor. For a windowed swap chain, the presentation is implemented by means of copy operations and always occurs immediately.

D3DSWAPEFFECT_FLIP

The swap chain might include multiple back buffers and is best envisaged as a circular queue that includes the front buffer. Within this queue, the back buffers are always numbered sequentially from 0 to (N -1), where N is the number of back buffers, so that 0 denotes the least recently presented buffer. When **Present** is invoked, the queue is "rotated" so that the front buffer becomes back buffer (N - 1), while the back buffer 0 becomes the new front buffer.

For a full-screen swap chain, the presentation rate is determined by the value assigned to the **FullScreen_PresentationInterval** member of the D3DPRESENT_PARAMETERS structure when the device or swap chain is created. Unless this value is D3DPRESENT_INTERVAL_IMMEDIATE, the presentation will be synchronized with the vertical sync of the monitor. For a windowed swap chain, the flipping is implemented by means of copy operations and the presentation always occurs immediately.

D3DSWAPEFFECT_COPY

This swap effect may be specified only for a swap chain comprising a single back buffer. Whether the swap chain is windowed or full-screen, the runtime will guarantee the semantics implied by a copy-based **Present** operation, namely that the operation leaves the content of the back buffer unchanged, instead of

replacing it with the content of the front buffer as a flip-based **Present** operation would.

For a windowed swap chain, a **Present** operation causes the back buffer content to be copied to the client area of the target window immediately. No attempt is made to synchronize the copy with the vertical retrace period of the display adapter, so "tearing" effects may be observed. In windowed applications you might want to use D3DSWAPEFFECT_COPY_VSYNC instead to eliminate, or at least minimize, such tearing effects.

For a full-screen swap chain, the runtime uses a combination of flip operations and copy operations, supported if necessary by hidden back buffers, to accomplish the **Present** operation. Accordingly, the presentation is synchronized with the display adapter's vertical retrace and its rate is constrained by the chosen presentation interval. A swap chain specified with the D3DPRESENT_INTERVAL_IMMEDIATE flag is the only exception. (Refer to the description of the **FullScreen_PresentationInterval** member of the **D3DPRESENT_PARAMETERS** structure.) In this case, a **Present** operation copies the back buffer content directly to the front buffer without waiting for the vertical retrace.

D3DSWAPEFFECT_COPY_VSYNC

Like D3DSWAPEFFECT_COPY, this swap effect may be used only with a swap chain made of a single back buffer. It guarantees that a **Present** operation applied to the swap chain will exhibit copy semantics, as described above for D3DSWAPEFFECT_COPY.

For a windowed swap chain, a **Present** operation causes the back buffer content to be copied to the client area of the target window. The runtime will attempt to eliminate tearing effects by avoiding the copy operation while the adapter is scanning within the destination rectangle on the display. It will also perform at most one such copy operation during the adapter's refresh period and thus limit the presentation frequency. Note, however, that if the adapter does not support the ability to report the raster status, the swap chain will behave as though it had been created with the D3DSWAPEFFECT_COPY swap effect. (Refer to the description of the D3DCAPS_READ_SCANLINE flag value for the **Caps** member of **D3DCAPS8**.)

For a full-screen swap chain, D3DSWAPEFFECT_COPY_VSYNC is identical to D3DSWAPEFFECT_COPY, except that the D3DPRESENT_INTERVAL_IMMEDIATE flag is meaningless when used in conjunction with D3DSWAPEFFECT_COPY_VSYNC. (Refer to the description of the **FullScreen_PresentationInterval** member of the **D3DPRESENT_PARAMETERS** structure.)

D3DSWAPEFFECT_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The state of the back buffer after a call to **Present** is well-defined by each of these swap effects, and whether the Microsoft® Direct3D® device was created with a full-

screen swap chain or a windowed swap chain has no effect on this state. In particular, the D3DSWAPEFFECT_FLIP swap effect operates the same whether windowed or full-screen, and the Direct3D runtime guarantees this by creating extra buffers. As a result, it is recommended that applications use D3DSWAPEFFECT_DISCARD whenever possible to avoid any such penalties. This is because this swap effect will always be the most efficient in terms of memory consumption and performance.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::Reset, **D3DPRESENT_PARAMETERS**

D3DTEXTUREADDRESS

Defines constants that describe the supported texture-addressing modes.

```
typedef enum _D3DTEXTUREADDRESS {
    D3DTADDRESS_WRAP           = 1,
    D3DTADDRESS_MIRROR         = 2,
    D3DTADDRESS_CLAMP          = 3,
    D3DTADDRESS_BORDER         = 4,
    D3DTADDRESS_MIRRORONCE     = 5,

    D3DTADDRESS_FORCE_DWORD    = 0x7fffffff
} D3DTEXTUREADDRESS;
```

Constants

D3DTADDRESS_WRAP
: Tile the texture at every integer junction. For example, for u values between 0 and 3, the texture is repeated three times; no mirroring is performed.

D3DTADDRESS_MIRROR
: Similar to D3DTADDRESS_WRAP, except that the texture is flipped at every integer junction. For u values between 0 and 1, for example, the texture is addressed normally; between 1 and 2, the texture is flipped (mirrored); between 2 and 3, the texture is normal again, and so on.

D3DTADDRESS_CLAMP
: Texture coordinates outside the range [0.0, 1.0] are set to the texture color at 0.0 or 1.0, respectively.

D3DTADDRESS_BORDER
: Texture coordinates outside the range [0.0, 1.0] are set to the border color.

D3DTADDRESS_MIRRORONCE

Similar to D3DTADDRESS_MIRROR and D3DTADDRESS_CLAMP. Takes the absolute value of the texture coordinate (thus, mirroring around 0), and then clamps to the maximum value. The most common usage is for volume textures, where support for the full D3DTADDRESS_MIRRORONCE texture-addressing mode is not necessary, but the data is symmetric around the one axis.

D3DTADDRESS_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DTEXTURESTAGESTATETYPE

D3DTEXTUREFILTERTYPE

Defines texture filtering modes for a texture stage.

```
typedef enum _D3DTEXTUREFILTERTYPE {
    D3DTEXF_NONE            = 0,
    D3DTEXF_POINT           = 1,
    D3DTEXF_LINEAR          = 2,
    D3DTEXF_ANISOTROPIC     = 3,
    D3DTEXF_FLATCUBIC       = 4,
    D3DTEXF_GAUSSIANCUBIC   = 5,

    D3DTEXF_FORCE_DWORD     = 0x7fffffff
} D3DTEXTUREFILTERTYPE;
```

Constants

D3DTEXF_NONE
Mipmapping disabled. The rasterizer should use the magnification filter instead.

D3DTEXF_POINT
Point filtering used as a texture magnification or minification filter. The texel with coordinates nearest to the desired pixel value is used.

The texture filter to be used between mipmap levels is nearest-point mipmap filtering. The rasterizer uses the color from the texel of the nearest mipmap texture.

D3DTEXF_LINEAR
Bilinear interpolation filtering used as a texture magnification or minification filter. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

The texture filter to use between mipmap levels is trilinear mipmap interpolation. The rasterizer linearly interpolates pixel color, using the texels of the two nearest mipmap textures.

D3DTEXF_ANISOTROPIC
Anisotropic texture filtering used as a texture magnification or minification filter. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

D3DTEXF_FLATCUBIC
Flat-cubic filtering used as a texture magnification filter.

D3DTEXF_GAUSSIANCUBIC
Gaussian-cubic filtering used as a texture magnification filter.

D3DTEXF_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Not all valid filtering modes for a device will apply to volume maps. In general, D3DTEXF_POINT and D3DTEXF_LINEAR magnification filters will be supported for volume maps. If D3DPTEXTURECAPS_MIPVOLUMEMAP is set, then the D3DTEXF_POINT mipmap filter and D3DTEXF_POINT and D3DTEXF_LINEAR minification filters will be supported for volume maps. The device may or may not support the D3DTEXF_LINEAR mipmap filter for volume maps. DirectX 8 devices that support anisotropic filtering for 2-D maps do not necessarily support anisotropic filtering for volume maps. However, applications that enable anisotropic filtering will receive the best available filtering (probably linear) if anisotropic filtering is not supported.

Set a texture stage's magnification filter by calling the **IDirect3DDevice8::SetTextureStageState** method with the D3DTSS_MAGFILTER value as the second parameter and one member of this enumeration as the third parameter.

Set a texture stage's minification filter by calling **SetTextureStageState** with the D3DTSS_MINFILTER value as the second parameter and one member of this enumeration as the third parameter.

Set the texture filter to use between mipmap levels by calling **SetTextureStageState** with the D3DTSS_MIPFILTER value as the second parameter and one member of this enumeration as the third parameter.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DTEXTURESTAGESTATETYPE

D3DTEXTUREOP

Defines per-stage texture-blending operations.

```
typedef enum _D3DTEXTUREOP {
    D3DTOP_DISABLE                = 1,
    D3DTOP_SELECTARG1             = 2,
    D3DTOP_SELECTARG2             = 3,
    D3DTOP_MODULATE               = 4,
    D3DTOP_MODULATE2X             = 5,
    D3DTOP_MODULATE4X             = 6,
    D3DTOP_ADD                    = 7,
    D3DTOP_ADDSIGNED              = 8,
    D3DTOP_ADDSIGNED2X            = 9,
    D3DTOP_SUBTRACT               = 10,
    D3DTOP_ADDSMOOTH              = 11,
    D3DTOP_BLENDDIFFUSEALPHA      = 12,
    D3DTOP_BLENDTEXTUREALPHA      = 13,
    D3DTOP_BLENDFACTORALPHA       = 14,
    D3DTOP_BLENDTEXTUREALPHAPM    = 15,
    D3DTOP_BLENDCURRENTALPHA      = 16,
    D3DTOP_PREMODULATE            = 17,
    D3DTOP_MODULATEALPHA_ADDCOLOR = 18,
    D3DTOP_MODULATECOLOR_ADDALPHA = 19,
    D3DTOP_MODULATEINVALPHA_ADDCOLOR = 20,
    D3DTOP_MODULATEINVCOLOR_ADDALPHA = 21,
    D3DTOP_BUMPENVMAP             = 22,
    D3DTOP_BUMPENVMAPLUMINANCE    = 23,
    D3DTOP_DOTPRODUCT3            = 24,
    D3DTOP_MULTIPLYADD            = 25,
    D3DTOP_LERP                   = 26,

    D3DTOP_FORCE_DWORD            = 0x7fffffff,
} D3DTEXTUREOP;
```

Constants

Control members

D3DTOP_DISABLE

Disables output from this texture stage and all stages with a higher index. To disable texture mapping, set this as the color operation for the first texture stage (stage 0). Alpha operations cannot be disabled when color operations are enabled. Setting the alpha operation to D3DTOP_DISABLE when color blending is enabled causes undefined behavior.

D3DTOP_SELECTARG1

Use this texture stage's first color or alpha argument, unmodified, as the output. This operation affects the color argument when used with the D3DTSS_COLOROP texture-stage state, and the alpha argument when used with D3DTSS_ALPHAOP.

$S_{RGBA} = Arg1$

D3DTOP_SELECTARG2

Use this texture stage's second color or alpha argument, unmodified, as the output. This operation affects the color argument when used with the D3DTSS_COLOROP texture stage state, and the alpha argument when used with D3DTSS_ALPHAOP.

$S_{RGBA} = Arg2$

Modulation members

D3DTOP_MODULATE

Multiply the components of the arguments.

$S_{RGBA} = Arg1 \times Arg2$

D3DTOP_MODULATE2X

Multiply the components of the arguments, and shift the products to the left 1 bit (effectively multiplying them by 2) for brightening.

$S_{RGBA} = (Arg1 \times Arg2) << 1$

D3DTOP_MODULATE4X

Multiply the components of the arguments, and shift the products to the left 2 bits (effectively multiplying them by 4) for brightening.

$S_{RGBA} = (Arg1 \times Arg2) << 2$

Addition and subtraction members

D3DTOP_ADD

Add the components of the arguments.

$S_{RGBA} = Arg1 + Arg2$

D3DTOP_ADDSIGNED

Add the components of the arguments with a –0.5 bias, making the effective range of values from –0.5 through 0.5.

$S_{RGBA} = Arg1 + Arg2 - 0.5$

D3DTOP_ADDSIGNED2X

Add the components of the arguments with a –0.5 bias, and shift the products to the left 1 bit.

$S_{RGBA} = (Arg1 + Arg2 - 0.5) << 1$

D3DTOP_SUBTRACT

Subtract the components of the second argument from those of the first argument.

$S_{RGBA} = Arg1 - Arg2$

D3DTOP_ADDSMOOTH

Add the first and second arguments; then subtract their product from the sum.

$S_{RGBA} = Arg\ 1 + Arg2 - Arg1 \times Arg2$
$= Arg1 + Arg2\ (1 - Arg1\ )$

Linear alpha blending members

D3DTOP_BLENDDIFFUSEALPHA,D3DTOP_BLENDTEXTUREALPHA, D3DTOP_BLENDFACTORALPHA, and D3DTOP_BLENDCURRENTALPHA

Linearly blend this texture stage, using the interpolated alpha from each vertex (D3DTOP_BLENDDIFFUSEALPHA), alpha from this stage's texture (D3DTOP_BLENDTEXTUREALPHA), a scalar alpha (D3DTOP_BLENDFACTORALPHA) set with the D3DRS_TEXTUREFACTOR render state, or the alpha taken from the previous texture stage (D3DTOP_BLENDCURRENTALPHA).

$S_{RGBA} = Arg\ 1 \times (Alpha) + Arg2 \times (1 - Alpha\ )$

D3DTOP_BLENDTEXTUREALPHAPM

Linearly blend a texture stage that uses a premultiplied alpha.

$S_{RGBA} = Arg\ 1 + Arg2 \times (1 - Alpha\ )$

Specular mapping members

D3DTOP_PREMODULATE
Modulate this texture stage with the next texture stage.

D3DTOP_MODULATEALPHA_ADDCOLOR
Modulate the color of the second argument, using the alpha of the first argument; then add the result to argument one. This operation is supported only for color operations (D3DTSS_COLOROP).

$$S_{RGBA} = Arg\ 1_{RGB} + Arg1_{A} \times Arg2_{RGB}$$

D3DTOP_MODULATECOLOR_ADDALPHA
Modulate the arguments; then add the alpha of the first argument. This operation is supported only for color operations (D3DTSS_COLOROP).

$$S_{RGBA} = Arg\ 1_{RGB} \times Arg2_{RGB} + Arg1_{A}$$

D3DTOP_MODULATEINVALPHA_ADDCOLOR
Similar to D3DTOP_MODULATEALPHA_ADDCOLOR, but use the inverse of the alpha of the first argument. This operation is supported only for color operations (D3DTSS_COLOROP).

$$S_{RGBA} = (1 - Arg\ 1_{A}) \times Arg2_{RGB} + Arg1_{RGB}$$

D3DTOP_MODULATEINVCOLOR_ADDALPHA
Similar to D3DTOP_MODULATECOLOR_ADDALPHA, but use the inverse of the color of the first argument. This operation is supported only for color operations (D3DTSS_COLOROP).

$$S_{RGBA} = (1 - Arg\ 1_{RGB}) \times Arg2_{RGB} + Arg1_{A}$$

Bump-mapping members

D3DTOP_BUMPENVMAP
Perform per-pixel bump mapping, using the environment map in the next texture stage, without luminance. This operation is supported only for color operations (D3DTSS_COLOROP).

D3DTOP_BUMPENVMAPLUMINANCE
Perform per-pixel bump mapping, using the environment map in the next texture stage, with luminance. This operation is supported only for color operations (D3DTSS_COLOROP).

D3DTOP_DOTPRODUCT3

Modulate the components of each argument as signed components, add their products; then replicate the sum to all color channels, including alpha. This operation is supported for color and alpha operations.

$$S_{RGBA} = (Arg\ 1_R \times Arg2_R + Arg1_G \times Arg2_G + Arg1_B \times Arg2_B)$$

In DirectX 6.0 and 7.0 multitexture operations the above inputs are all shifted down by half (y = x - 0.5) before use to simulate signed data, and the scalar result is automatically clamped to positive values and replicated to all three output channels. Also, note that as a color operation this does not updated the alpha it just updates the rgb components.

However, in DirectX 8.0 shaders you can specify that the output be routed to the .rgb or the .a components or both (the default). You can also specify a separate scalar operation on the alpha channel.

Triadic texture blending members

D3DTOP_MULTIPLYADD

Performs a multiply-accumulate operation. It takes the last two arguments, multiplies them together, and adds them to the remaining input/source argument, and places that into the result.

$S_{RGBA} = Arg1 + Arg2 * Arg3$

D3DTOP_LERP

Linearly interpolates between the 2nd and 3rd source arguments by a proportion specified in the 1st source argument.

$S_{RGBA} = (Arg1) * Arg2 + (1\text{-}Arg1) * Arg3.$

Miscellaneous member

D3DTOP_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The members of this type are used when setting color or alpha operations by using the D3DTSS_COLOROP or D3DTSS_ALPHAOP values with the **IDirect3DDevice8::SetTextureStageState** method.

In the above formulas, S_{RGBA} is the RGBA color produced by a texture operation, and *Arg1*, *Arg2*, and *Arg3* represent the complete RGBA color of the texture arguments. Individual components of an argument are shown with subscripts. For example, the alpha component for argument 1 would be shown as $Arg1_A$.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DTEXTURESTAGESTATETYPE

D3DTEXTURESTAGESTATETYPE

Defines texture stage states.

```
typedef enum _D3DTEXTURESTAGESTATETYPE {
    D3DTSS_COLOROP            =  1,
    D3DTSS_COLORARG1          =  2,
    D3DTSS_COLORARG2          =  3,
    D3DTSS_ALPHAOP            =  4,
    D3DTSS_ALPHAARG1          =  5,
    D3DTSS_ALPHAARG2          =  6,
    D3DTSS_BUMPENVMAT00       =  7,
    D3DTSS_BUMPENVMAT01       =  8,
    D3DTSS_BUMPENVMAT10       =  9,
    D3DTSS_BUMPENVMAT11       = 10,
    D3DTSS_TEXCOORDINDEX      = 11,
    D3DTSS_ADDRESSU           = 13,
    D3DTSS_ADDRESSV           = 14,
    D3DTSS_BORDERCOLOR        = 15,
    D3DTSS_MAGFILTER          = 16,
    D3DTSS_MINFILTER          = 17,
    D3DTSS_MIPFILTER          = 18,
    D3DTSS_MIPMAPLODBIAS      = 19,
    D3DTSS_MAXMIPLEVEL        = 20,
    D3DTSS_MAXANISOTROPY      = 21,
    D3DTSS_BUMPENVLSCALE      = 22,
    D3DTSS_BUMPENVLOFFSET     = 23,
    D3DTSS_TEXTURETRANSFORMFLAGS = 24,
    D3DTSS_ADDRESSW           = 25,
    D3DTSS_COLORARG0          = 26,
    D3DTSS_ALPHAARG0          = 27,
    D3DTSS_RESULTARG          = 28,

    D3DTSS_FORCE_DWORD        = 0x7fffffff
} D3DTEXTURESTAGESTATETYPE;
```

Constants

D3DTSS_COLOROP

The texture-stage state is a texture color blending operation identified by one member of the **D3DTEXTUREOP** enumerated type. The default value for the

first texture stage (stage 0) is D3DTOP_MODULATE, and for all other stages the default is D3DTOP_DISABLE.

D3DTSS_COLORARG1

The texture-stage state is the first color argument for the stage, identified by a texture argument flag. The default argument is D3DTA_TEXTURE.

Specify D3DTA_TEMP to select a temporary register color for read or write. D3DTA_TEMP is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present. The default value for the register is (0.0, 0.0, 0.0, 0.0)

D3DTSS_COLORARG2

The texture-stage state is the second color argument for the stage, identified by a texture argument flag. The default argument is D3DTA_CURRENT.

Specify D3DTA_TEMP to select a temporary register color for read or write. D3DTA_TEMP is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present. The default value for the register is (0.0, 0.0, 0.0, 0.0)

D3DTSS_ALPHAOP

The texture-stage state is a texture alpha blending operation identified by one member of the **D3DTEXTUREOP** enumerated type. The default value for the first texture stage (stage 0) is D3DTOP_SELECTARG1, and for all other stages the default is D3DTOP_DISABLE.

D3DTSS_ALPHAARG1

The texture-stage state is the first alpha argument for the stage, identified by a texture argument flag. The default argument is D3DTA_TEXTURE. If no texture is set for this stage, the default argument is D3DTA_DIFFUSE.

Specify D3DTA_TEMP to select a temporary register color for read or write. D3DTA_TEMP is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present. The default value for the register is (0.0, 0.0, 0.0, 0.0)

D3DTSS_ALPHAARG2

The texture-stage state is the second alpha argument for the stage, identified by a texture argument flag. The default argument is D3DTA_CURRENT.

Specify D3DTA_TEMP to select a temporary register color for read or write. D3DTA_TEMP is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present. The default value for the register is (0.0, 0.0, 0.0, 0.0)

D3DTSS_BUMPENVMAT00

The texture-stage state is a floating-point value for the [0][0] coefficient in a bump-mapping matrix. The default value is 0.0.

D3DTSS_BUMPENVMAT01

The texture-stage state is a floating-point value for the [0][1] coefficient in a bump-mapping matrix. The default value is 0.0.

D3DTSS_BUMPENVMAT10

The texture-stage state is a floating-point value for the [1][0] coefficient in a bump-mapping matrix. The default value is 0.0.

D3DTSS_BUMPENVMAT11

The texture-stage state is a floating-point value for the [1][1] coefficient in a bump-mapping matrix. The default value is 0.0.

D3DTSS_TEXCOORDINDEX

Index of the texture coordinate set to use with this texture stage. This flag is used only for fixed-function vertex processing. For example, it should not be used with vertex shaders. When rendering using vertex shaders, each stage's texture coordinate index must be set to its default value. The default index for each stage is equal to the stage index. Set this state to the zero-based index of the coordinate set for each vertex that this texture stage uses. You can specify up to eight sets of texture coordinates per vertex. If a vertex does not include a set of texture coordinates at the specified index, the system defaults to the u and v coordinates (0,0).

Additionally, applications can include, as logical **Or** with the index being set, one of the following flags to request that Microsoft® Direct3D® automatically generate the input texture coordinates for a texture transformation. With the exception of D3DTSS_TCI_PASSTHRU, which resolves to zero, if any of the following values is included with the index being set, the system uses the index strictly to determine texture wrapping mode. These flags are most useful when performing environment mapping.

D3DTSS_TCI_PASSTHRU

Use the specified texture coordinates contained within the vertex format. This value resolves to zero.

D3DTSS_TCI_CAMERASPACENORMAL

Use the vertex normal, transformed to camera space, as the input texture coordinates for this stage's texture transformation.

D3DTSS_TCI_CAMERASPACEPOSITION

Use the vertex position, transformed to camera space, as the input texture coordinates for this stage's texture transformation.

D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR

Use the reflection vector, transformed to camera space, as the input texture coordinate for this stage's texture transformation. The reflection vector is computed from the input vertex position and normal vector.

D3DTSS_ADDRESSU

Member of the **D3DTEXTUREADDRESS** enumerated type. Selects the texture-addressing method for the u coordinate. The default is D3DTADDRESS_WRAP.

D3DTSS_ADDRESSV

Member of the **D3DTEXTUREADDRESS** enumerated type. Selects the texture-addressing method for the v coordinate. The default value is D3DTADDRESS_WRAP.

D3DTSS_BORDERCOLOR

D3DCOLOR value that describes the color to use for rasterizing texture coordinates outside the [0.0,1.0] range. The default color is 0x00000000.

D3DTSS_MAGFILTER

Member of the **D3DTEXTUREFILTERTYPE** enumerated type that indicates the texture magnification filter to use when rendering the texture onto primitives. The default value is D3DTEXF_POINT.

D3DTSS_MINFILTER
Member of the **D3DTEXTUREFILTERTYPE** enumerated type that indicates the texture minification filter to use when rendering the texture onto primitives. The default value is D3DTEXF_POINT.

D3DTSS_MIPFILTER
Member of the **D3DTEXTUREFILTERTYPE** enumerated type that indicates the texture filter to use between mipmap levels. The default value is D3DTEXF_NONE.

D3DTSS_MIPMAPLODBIAS
Level of detail bias for mipmaps. Can be used to make textures appear more chunky or more blurred. The default value is zero. Values for this state are floating-point values. Because the **IDirect3DDevice8::SetTextureStageState** and **IDirect3DDevice8::GetTextureStageState** accept **DWORD** values, your application must cast a variable that contains the value, as shown in the following code example.

```
pd3dDevice->SetTextureStageState(D3DTSS_MIPMAPLODBIAS, *((LPDWORD)
(&fBias)));
```

Each unit of bias (+/-1.0) alters the selection by exactly one mipmap level. A negative bias causes the use of larger mipmap levels; the result is a sharper but more aliased image. A positive bias causes the use of smaller mipmap levels; the result is a more blurred image. A positive bias also causes less texture data to be referenced, which can boost performance on some systems.

D3DTSS_MAXMIPLEVEL
Maximum mipmap level of detail that the application allows, expressed as an index from the top of the mipmap chain. Lower values identify higher levels of detail within the mipmap chain. Zero, which is the default, indicates that all levels can be used. Nonzero values indicate that the application cannot display mipmaps that have a higher level of detail than the mipmap at the specified index.

D3DTSS_MAXANISOTROPY
Maximum level of anisotropy. The default value is 1.

D3DTSS_BUMPENVLSCALE
Floating-point scale value for bump-map luminance. The default value is 0.0.

D3DTSS_BUMPENVLOFFSET
Floating-point offset value for bump-map luminance. The default value is 0.0.

D3DTSS_TEXTURETRANSFORMFLAGS
Member of the **D3DTEXTURETRANSFORMFLAGS** enumerated type that controls the transformation of texture coordinates for this texture stage. The default value is D3DTTFF_DISABLE.

D3DTSS_ADDRESSW
Member of the **D3DTEXTUREADDRESS** enumerated type. Selects the texture-addressing method for the w coordinate. The default value is D3DTADDRESS_WRAP.

D3DTSS_COLORARG0

Settings for the third color operand for triadic operations (multiply add and linear interpolation), identified by a texture argument flag. This setting is supported if the D3DTEXOPCAPS_MULTIPLYADD or D3DTEXOPCAPS_LERP device capabilities are present.

Specify D3DTA_TEMP to select a temporary register color for read or write. D3DTA_TEMP is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present. The default value for the register is (0.0, 0.0, 0.0, 0.0)

D3DTSS_ALPHAARG0

Settings for the alpha channel selector operand for triadic operations (multiply add and linear interpolation), identified by a texture argument flag. This setting is supported if the D3DTEXOPCAPS_MULTIPLYADD or D3DTEXOPCAPS_LERP device capabilities are present.

Specify D3DTA_TEMP to select a temporary register color for read or write. D3DTA_TEMP is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present. The default value for the register is (0.0, 0.0, 0.0, 0.0)

D3DTSS_RESULTARG

Setting to select destination register for the result of this stage, identified by a texture argument flag. This value can be set to D3DTA_CURRENT (the default value) or to D3DTA_TEMP, which is a single temporary register that can be read into subsequent stages as an input argument. The final color passed to the fog blender and frame buffer is taken from D3DTA_CURRENT, so the last active texture stage state must be set to write to current.

This setting is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present.

D3DTSS_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Members of this enumerated type are used with the **IDirect3DDevice8::GetTextureStageState** and **IDirect3DDevice8::SetTextureStageState** methods to retrieve and set texture state values.

The valid range of values for the D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT01, D3DTSS_BUMPENVMAT10, and D3DTSS_BUMPENVMAT11 bump-mapping matrix coefficients is greater than or equal to –8.0 and less than 8.0. This range, expressed in mathematical notation is [–8.0,8.0).

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetTextureStageState,
IDirect3DDevice8::SetTextureStageState

D3DTEXTURETRANSFORMFLAGS

Defines texture-stage state values.

```
typedef enum _D3DTEXTURETRANSFORMFLAGS {
    D3DTTFF_DISABLE        =   0,
    D3DTTFF_COUNT1         =   1,
    D3DTTFF_COUNT2         =   2,
    D3DTTFF_COUNT3         =   3,
    D3DTTFF_COUNT4         =   4,
    D3DTTFF_PROJECTED      = 256,

    D3DTTFF_FORCE_DWORD    = 0x7fffffff
} D3DTEXTURETRANSFORMFLAGS;
```

Constants

D3DTTFF_DISABLE
: Texture coordinates are passed directly to the rasterizer.

D3DTTFF_COUNT1
: The rasterizer should expect 1-D texture coordinates.

D3DTTFF_COUNT2
: The rasterizer should expect 2-D texture coordinates.

D3DTTFF_COUNT3
: The rasterizer should expect 3-D texture coordinates.

D3DTTFF_COUNT4
: The rasterizer should expect 4-D texture coordinates.

D3DTTFF_PROJECTED
: The texture coordinates are all divided by the last element before being passed to the rasterizer. For example, if this flag is specified with the D3DTTFF_COUNT3 flag, the first and second texture coordinates is divided by the third coordinate before being passed to the rasterizer.

D3DTTFF_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

These flags are used to set the value of the D3DTSS_TEXTURETRANSFORMFLAGS texture stage state for the **D3DTEXTURESTAGESTATETYPE** enumerated type.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DTEXTURESTAGESTATETYPE

D3DTRANSFORMSTATETYPE

Defines constants that describe transformation state values.

```
typedef enum _D3DTRANSFORMSTATETYPE {
    D3DTS_VIEW          =  2,
    D3DTS_PROJECTION    =  3,
    D3DTS_TEXTURE0      = 16,
    D3DTS_TEXTURE1      = 17,
    D3DTS_TEXTURE2      = 18,
    D3DTS_TEXTURE3      = 19,
    D3DTS_TEXTURE4      = 20,
    D3DTS_TEXTURE5      = 21,
    D3DTS_TEXTURE6      = 22,
    D3DTS_TEXTURE7      = 23,

    D3DTS_FORCE_DWORD   = 0x7fffffff
} D3DTRANSFORMSTATETYPE;
```

Constants

D3DTS_VIEW
: Identifies the transformation matrix being set as the view transformation matrix. The default value is NULL (the identity matrix).

D3DTS_PROJECTION
: Identifies the transformation matrix being set as the projection transformation matrix. The default value is NULL (the identity matrix).

D3DTS_TEXTURE0 through D3DTS_TEXTURE7
: Identifies the transformation matrix being set for the specified texture stage.

D3DTS_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size. This value is not used.

Macros

D3DTS_WORLD
: Equivalent to D3DTS_WORLDMATRIX(0).

D3DTS_WORLDMATRIX(*index*)

Identifies the transform matrix to set for the world matrix at *index*. Multiple world matrices are used only for Vertex Blending. Otherwise only D3DTS_WORLD is used.

Remarks

The transform states in the range 256 through 511 are reserved to store up to 256 world matrices that can be indexed using the D3DTS_WORLDMATRIX and D3DTS_WORLD macros.

Requirements

Header: Declared in D3d8types.h.

See Also

IDirect3DDevice8::GetTransform, **IDirect3DDevice8::MultiplyTransform**, **IDirect3DDevice8::SetTransform**, **D3DTS_WORLD**, **D3DTS_WORLD***n*, **D3DTS_WORLDMATRIX**

D3DVERTEXBLENDFLAGS

Defines flags used to control the number or matrices that the system applies when performing multimatrix vertex blending.

```
typedef enum _D3DVERTEXBLENDFLAGS {
    D3DVBF_DISABLE  =   0,
    D3DVBF_1WEIGHTS =   1,
    D3DVBF_2WEIGHTS =   2,
    D3DVBF_3WEIGHTS =   3,
    D3DVBF_TWEENING = 255,
    D3DVBF_0WEIGHTS = 256
} D3DVERTEXBLENDFLAGS;
```

Constants

D3DVBF_DISABLE
: Disable vertex blending; apply only the world matrix set by the **D3DTS_WORLDMATRIX** macro, where the index value for the transformation state is 0.

D3DVBF_1WEIGHTS
: Enable vertex blending between the two matrices set by the **D3DTS_WORLDMATRIX** macro, where the index value for the transformation states are 0 and 1.

D3DVBF_2WEIGHTS

Enable vertex blending between the three matrices set by the **D3DTS_WORLDMATRIX** macro, where the index value for the transformation states are 0, 1, and 2.

D3DVBF_3WEIGHTS
Enable vertex blending between the four matrices set by the **D3DTS_WORLDMATRIX** macro, where the index value for the transformation states are 0, 1, 2, and 3.

D3DVBF_TWEENING
Vertex blending is done by using the value assigned to D3DRS_TWEENFACTOR.

D3DVBF_0WEIGHTS
Use a single matrix with a weight of 1.0.

Remarks

Members of this type are used with the D3DRS_VERTEXBLEND render state.

Geometry blending (multimatrix vertex blending) requires that your application use a vertex format that has blending (beta) weights for each vertex.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE, **D3DTS_WORLD**, **D3DTS_WORLD*n***, **D3DTS_WORLDMATRIX**,

D3DZBUFFERTYPE

Defines constants that describe depth-buffer formats.

```
typedef enum _D3DZBUFFERTYPE {
    D3DZB_FALSE             = 0,
    D3DZB_TRUE              = 1,
    D3DZB_USEW              = 2,

    D3DZB_FORCE_DWORD       = 0x7fffffff
} D3DZBUFFERTYPE;
```

Constants

D3DZB_FALSE
Disable depth buffering.

D3DZB_TRUE

Enable z-buffering.

D3DZB_USEW
Enable w-buffering.

D3DZB_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Members of this enumerated type are used with the D3DRS_ZENABLE render state.

Requirements

Header: Declared in D3d8types.h.

See Also

D3DRENDERSTATETYPE

Other Types

This section contains information about the following Microsoft® Direct3D® type that is neither a structure nor enumerated type.

- **D3DCOLOR**

D3DCOLOR

Defines the fundamental Microsoft® Direct3D® color type.

```
typedef DWORD D3DCOLOR;
```

Requirements

Header: Declared in D3d8types.h.

Texture Argument Flags

Each texture stage for a device can have two texture arguments that affect the color or alpha channel of the texture.

Set and retrieve texture arguments by calling the **IDirect3DDevice8::SetTextureStageState** and **IDirect3DDevice8::GetTextureStageState** methods, and specifying the D3DTSS_COLORARG0, D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAARG0, D3DTSS_ALPHAARG1, D3DTSS_ALPHAARG2, or

D3DTSS_RESULTARG member of the **D3DTEXTURESTAGESTATETYPE** enumerated type.

The following flags, organized as arguments and modifiers, can be used with color and alpha arguments for a texture stage. You can combine an argument flag with a modifier, but two argument flags cannot be combined.

Argument flags

D3DTA_CURRENT
: The texture argument is the result of the previous blending stage. In the first texture stage (stage 0), this argument is equivalent to D3DTA_DIFFUSE. If the previous blending stage uses a bump-map texture (the D3DTOP_BUMPENVMAP operation), the system chooses the texture from the stage before the bump-map texture. If *s* represents the current texture stage and *s* - 1 contains a bump-map texture, this argument becomes the result output by texture stage *s* - 2. Permissions are read/write.

D3DTA_DIFFUSE
: The texture argument is the diffuse color interpolated from vertex components during Gouraud shading. If the vertex does not contain a diffuse color, the default color is 0xFFFFFFFF. Permissions are read-only.

D3DTA_SELECTMASK
: Mask value for all arguments; not used when setting texture arguments.

D3DTA_SPECULAR
: The texture argument is the specular color interpolated from vertex components during Gouraud shading. If the vertex does not contain a specular color, the default color is 0xFFFFFFFF. Permissions are read only.

D3DTA_TEMP
: The texture argument is a temporary register color for read or write. D3DTA_TEMP is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present. The default value for the register is (0.0, 0.0, 0.0, 0.0). Permissions are read/write.

D3DTA_TEXTURE
: The texture argument is the texture color for this texture stage. Permissions are read-only.

D3DTA_TFACTOR
: The texture argument is the texture factor set in a previous call to the **IDirect3DDevice8::SetRenderState** with the D3DRS_TEXTUREFACTOR render-state value. Permissions are read-only.

Modifier flags

D3DTA_ALPHAREPLICATE
: Replicate the alpha information to all color channels before the operation completes. This is a read modifier.

D3DTA_COMPLEMENT
: Invert the argument so that, if the result of the argument were referred to by the variable *x*, the value would be 1.0 minus *x*. This is a read modifier.

Flexible Vertex Format Flags

The flexible vertex format (FVF) is used to describe the contents of vertices stored interleaved in a single data stream. A FVF code is generally used to specify data to be processed by fixed function vertex processing.

The following flags describe a vertex format. For information regarding vertex formats, see About Vertex Formats.

Flexible vertex format (FVF) flags

D3DFVF_DIFFUSE
: Vertex format includes a diffuse color component.

D3DFVF_NORMAL
: Vertex format includes a vertex normal vector. This flag cannot be used with the D3DFVF_XYZRHW flag.

D3DFVF_PSIZE
: Vertex format specified in point size. This size is expressed in camera space units for vertices that are not transformed and lit, and in device-space units for transformed and lit vertices.

D3DFVF_SPECULAR
: Vertex format includes a specular color component.

D3DFVF_XYZ
: Vertex format includes the position of an untransformed vertex. This flag cannot be used with the D3DFVF_XYZRHW flag.

D3DFVF_XYZRHW
: Vertex format includes the position of a transformed vertex. This flag cannot be used with the D3DFVF_XYZ or D3DFVF_NORMAL flags.

D3DFVF_XYZB1 through D3DFVF_XYZB5
: Vertex format contains position data, and a corresponding number of weighting (beta) values to use for multimatrix vertex blending operations. Currently, Microsoft® Direct3D® can blend with up to three weighting values and four blending matrices. For more information on using blending matrices, see Indexed Vertex Blending.

Texture-related FVF flags

D3DFVF_TEX0 through D3DFVF_TEX8
: Number of texture coordinate sets for this vertex. The actual values for these flags are not sequential.

D3DFVF_TEXTUREFORMAT1 through D3DFVF_TEXTUREFORMAT4
: Number of values that define a texture coordinate set. The D3DFVF_TEXTUREFORMAT1 indicates one-dimensional texture coordinates, D3DFVF_TEXTUREFORMAT2 indicates two-dimensional texture coordinates, and so on. These flags are rarely used alone; they are used with the **D3DFVF_TEXCOORDSIZE*n*** macros.

Mask values

D3DFVF_POSITION_MASK
: Mask for position bits.

D3DFVF_RESERVED0 and D3DFVF_RESERVED2
Mask values for reserved bits in the flexible vertex format. Do not use.

D3DFVF_TEXCOUNT_MASK
Mask value for texture flag bits.

Miscellaneous

D3DFVF_LASTBETA_UBYTE4
When using indexed vertex blending and a fixed function FVF vertex shader, you must specify this flag for the vertex shader.

D3DFVF_TEXCOUNT_SHIFT
The number of bits by which to shift an integer value that identifies the number of a texture coordinates for a vertex. This value might be used as follows:

```
DWORD dwNumTextures = 1;  // Vertex has only one set of
                                        // coordinates.

// Shift the value for use when creating an FVF combination.
dwFVF = dwNumTextures<<D3DFVF_TEXCOUNT_SHIFT;

/*
 * Now, create an FVF combination using the shifted value.
 */
```

The following example shows some other common flag combinations.

```
// Lightweight, untransformed vertex for lit, untextured,
// Gouraud-shaded content.
dwFVF = ( D3DFVF_XYZ | D3DFVF_DIFFUSE );

// Untransformed vertex for unlit, untextured, Gouraud-shaded
// content with diffuse material color specified per vertex.
dwFVF = ( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE );

// Untransformed vertex for light-map-based lighting.
dwFVF = ( D3DFVF_XYZ | D3DFVF_TEX2 );

// Transformed vertex for light-map-based lighting
// with shared rhw.
dwFVF = ( D3DFVF_XYZRHW | D3DFVF_TEX2 );

// Heavyweight vertex for unlit, colored content with two
// sets of texture coordinates.
dwFVF = ( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE |
        D3DFVF_SPECULAR | D3DFVF_TEX2 );
```

See Also

About Vertex Formats, Geometry Blending

Four-Character Codes (FOURCC)

Microsoft® Direct3D® and the Direct3DX utility library use a special set of codes that are four characters in length. These codes, called four-character codes or FOURCCs, are stored in file headers of files that contain multimedia data, such as bitmap images, sound, or video. FOURCCs describe the software technology that was used to produce multimedia data. By implication, they also describe the format of the data itself.

Direct3D applications use FOURCCs for image color and format conversion.

FOURCCs are registered with Microsoft by the vendors of the respective multimedia software technologies. Some common FOURCCs appear in the following list.

FOURCC	Company	Technology name
AUR2	AuraVision Corporation	AuraVision Aura 2: YUV 422
AURA	AuraVision Corporation	AuraVision Aura 1: YUV 411
CHAM	Winnov, Inc.	MM_WINNOV_CAVIARA_CHAMPAGNE
CVID	Supermac	Cinepak by Supermac
CYUV	Creative Labs, Inc.	Creative Labs YUV
DXT1	Microsoft Corporation	DirectX Texture Compression Format 1
DXT2	Microsoft Corporation	DirectX Texture Compression Format 2
DXT3	Microsoft Corporation	DirectX Texture Compression Format 3
DXT4	Microsoft Corporation	DirectX Texture Compression Format 4
DXT5	Microsoft Corporation	DirectX Texture Compression Format 5
FVF1	Iterated Systems, Inc.	Fractal Video Frame
IF09	Intel Corporation	Intel Intermediate YUV9
IV31	Intel Corporation	Indeo 3.1
JPEG	Microsoft Corporation	Still Image JPEG DIB
MJPG	Microsoft Corporation	Motion JPEG DIB Format
MRLE	Microsoft Corporation	Run Length Encoding
MSVC	Microsoft Corporation	Video 1
PHMO	IBM Corporation	Photomotion
RT21	Intel Corporation	Indeo 2.1
ULTI	IBM Corporation	Ultimotion
V422	Vitec Multimedia	24-bit YUV 4:2:2
V655	Vitec Multimedia	16-bit YUV 4:2:2
VDCT	Vitec Multimedia	Video Maker Pro DIB

VIDS	Vitec Multimedia	YUV 4:2:2 CCIR 601 for V422
YU92	Intel Corporation	YUV
YUV8	Winnov, Inc.	MM_WINNOV_CAVIAR_YUV8
YUV9	Intel Corporation	YUV9
YUYV	Canopus, Co., Ltd.	BI_YUYV, Canopus
ZPEG	Metheus	Video Zipper

Return Values

Errors are represented by negative values and cannot be combined. The following lists the values that can be returned by Microsoft® Direct3D® methods. See the individual method descriptions for lists of the values that each can return. These lists are not necessarily comprehensive.

D3D_OK
No error occurred.

D3DERR_CONFLICTINGRENDERSTATE
The currently set render states cannot be used together.

D3DERR_CONFLICTINGTEXTUREFILTER
The current texture filters cannot be used together.

D3DERR_CONFLICTINGTEXTUREPALETTE
The current textures cannot be used simultaneously. This generally occurs when a multitexture device requires that all palletized textures simultaneously enabled also share the same palette.

D3DERR_DEVICELOST
The device is lost and cannot be restored at the current time, so rendering is not possible.

D3DERR_DEVICENOTRESET
The device cannot be reset.

D3DERR_DRIVERINTERNALERROR
Internal driver error.

D3DERR_INVALIDCALL
The method call is invalid. For example, a method's parameter may have an invalid value.

D3DERR_INVALIDDEVICE
The requested device type is not valid.

D3DERR_MOREDATA
There is more data available than the specified buffer size can hold.

D3DERR_NOTAVAILABLE
This device does not support the queried technique.

D3DERR_NOTFOUND

The requested item was not found.

D3DERR_OUTOFVIDEOMEMORY
Direct3D does not have enough display memory to perform the operation.

D3DERR_TOOMANYOPERATIONS
The application is requesting more texture-filtering operations than the device supports.

D3DERR_UNSUPPORTEDALPHAARG
The device does not support a specified texture-blending argument for the alpha channel.

D3DERR_UNSUPPORTEDALPHAOPERATION
The device does not support a specified texture-blending operation for the alpha channel.

D3DERR_UNSUPPORTEDCOLORARG
The device does not support a specified texture-blending argument for color values.

D3DERR_UNSUPPORTEDCOLOROPERATION
The device does not support a specified texture-blending operation for color values.

D3DERR_UNSUPPORTEDFACTORVALUE
The device does not support the specified texture factor value.

D3DERR_UNSUPPORTEDTEXTUREFILTER
The device does not support the specified texture filter.

D3DERR_WRONGTEXTUREFORMAT
The pixel format of the texture surface is not valid.

E_FAIL
An undetermined error occurred inside the Direct3D subsystem.

E_INVALIDARG
An invalid parameter was passed to the returning function

E_INVALIDCALL
The method call is invalid. For example, a method's parameter may have an invalid value.

E_OUTOFMEMORY
Direct3D could not allocate sufficient memory to complete the call.

S_OK
No error occurred.

Direct3DX C/C++ Reference

This section contains reference information for the API elements provided by the Direct3DX utility library. Reference material is divided into the following categories.

- Interfaces

- Functions
- Macros
- Structures
- C++ Specific Features
- Return Values

Interfaces

This section contains reference information for the COM interfaces provided by the Direct3DX utility library. The following interfaces are used with the Direct3DX utility library.

- **ID3DXBaseMesh**
- **ID3DXBuffer**
- **ID3DXEffect**
- **ID3DXFont**
- **ID3DXMatrixStack**
- **ID3DXMesh**
- **ID3DXPMesh**
- **ID3DXRenderToSurface**
- **ID3DXSkinMesh**
- **ID3DXSPMesh**
- **ID3DXSprite**
- **ID3DXTechnique**

ID3DXBaseMesh

Applications use the methods of the **ID3DXBaseMesh** interface to manipulate and query mesh and progressive mesh objects.

The methods of the **ID3DXBaseMesh** interface can be organized into the following groups.

Buffers	**GetIndexBuffer**
	GetVertexBuffer
	LockIndexBuffer
	LockVertexBuffer
	UnlockIndexBuffer
	UnlockVertexBuffer

Copying	**CloneMesh**
	CloneMeshFVF
Faces	**GetNumFaces**
Information	**GetDevice**
	GetOptions
Rendering	**DrawSubset**
	GetAttributeTable
Vertices	**GetDeclaration**
	GetFVF
	GetNumVertices

The **ID3DXBaseMesh** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

A mesh is an object made up of a set of polygonal faces. A mesh defines a set of vertices and a set of faces (the faces are defined in terms of the vertices and normals of the mesh).

The **LPD3DXBASEMESH** type is defined as a pointer to the **ID3DXBaseMesh** interface.

```
typedef struct ID3DXBaseMesh *LPD3DXBASEMESH;
```

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMesh, **ID3DXPMesh**

ID3DXBaseMesh::CloneMesh

Clones a mesh using a declarator.

```
HRESULT CloneMesh(
  DWORD Options,
  CONST DWORD* pDeclaration,
```

LPDIRECT3DDEVICE8 *pD3DDevice*,
LPD3DXMESH* *ppCloneMesh*
);

Parameters

Options

[in] A combination of one or more flags specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED

Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers. These meshes will have separate index buffers into the shared vertex buffer.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pDeclaration
[in] Pointer to an array of **DWORD** values, representing the declarator to describe the vertex format of the vertices in the output mesh. This parameter must map directly to an FVF.

pD3DDevice
[in] Pointer to a **IDirect3DDevice8** interface representing the device object associated with the mesh.

ppCloneMesh
[out, retval] Address of a pointer to an **ID3DXMesh** interface, representing the cloned mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::GetDeclaration, **D3DXDeclaratorFromFVF**

ID3DXBaseMesh::CloneMeshFVF

Clones a mesh using a flexible vertex format (FVF) code.

```
HRESULT CloneMeshFVF(
  DWORD Options,
  DWORD FVF,
  LPDIRECT3DDEVICE8 pD3DDevice,
  LPD3DXMESH* ppCloneMesh
);
```

Parameters

Options

[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
: The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
: Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
: Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
: Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
: Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
: Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
: Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
: Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
: Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers. These meshes will have separate index buffers into the shared vertex buffer.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

FVF
[in] Combination of flexible vertex format flags that specifies the vertex format for the vertices in the output mesh.

pD3DDevice
[in] Pointer to a **IDirect3DDevice8** interface representing the device object associated with the mesh.

ppCloneMesh
[out, retval] Address of a pointer to an **ID3DXMesh** interface, representing the cloned mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Remarks

CloneMeshFVF can be used to convert a mesh from one FVF to another.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::GetFVF, **D3DXFVFFromDeclarator**

ID3DXBaseMesh::DrawSubset

Draws a subset of a mesh.

```
HRESULT DrawSubset(
  DWORD AttribId
);
```

Parameters

AttribId
[in] **DWORD** that specifies which subset of the mesh to draw. This value is used to differentiate faces in a mesh as belonging to one or more attribute groups.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Remarks

An attribute table is used to identify areas of the mesh that need to be drawn with different textures, render states, materials, and so on. In addition, the application can use the attribute table to hide portions of a mesh by not drawing a given attribute identifier (*AttribId*) when drawing the frame.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::GetAttributeTable

ID3DXBaseMesh::GetAttributeTable

Retrieves either an attribute table for a mesh, or the number of entries stored in an attribute table for a mesh.

```
HRESULT GetAttributeTable(
  D3DXATTRIBUTERANGE* pAttribTable,
  DWORD* pAttribTableSize
);
```

Parameters

pAttribTable
[in, out] Pointer to an array of **D3DXATTRIBUTERANGE** structures, representing the entries in the mesh's attribute table. Specify NULL to retrieve the value for *pAttribTableSize*.

pAttribTableSize
[in, out] Pointer to either the number of entries stored in *pAttribTable* or a value to be filled in with the number of entries stored in the attribute table for the mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**

- **ID3DXPMesh**

Remarks

An attribute table is created by **ID3DXMesh::Optimize** and passing D3DXMESHOPT_ATTRSORT for the *Flags* parameter.

An attribute table is used to identify areas of the mesh that need to be drawn with different textures, render states, materials, and so on. In addition, the application can use the attribute table to hide portions of a mesh by not drawing a given attribute identifier when drawing the frame.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXBaseMesh::GetDeclaration

Retrieves a declaration describing the vertices in the mesh.

```
HRESULT GetDeclaration(
  DWORD Declaration[MAX_FVF_DECL_SIZE]
);
```

Parameters

Declaration
[in, out] A returned array describing the vertex format of the vertices in the queried mesh. The upper limit of this declarator array is MAX_FVF_DECL_SIZE, limiting the declarator to a maximum of 15 **DWORD**s.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::GetFVF

ID3DXBaseMesh::GetDevice

Retrieves the device associated with the mesh.

```
HRESULT GetDevice(
  LPDIRECT3DDEVICE8* ppDevice
```

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface, representing the Microsoft® Direct3D® device object associated with the mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Note

Calling this method will increase the internal reference count on the **IDirect3DDevice8** interface. Be sure to call **IUnknown::Release** when you are done using this **IDirect3DDevice8** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXBaseMesh::GetFVF

Retrieves the flexible vertex format of the vertices in the mesh.

```
DWORD GetFVF();
```

Parameters

None.

Return Values

Returns a combination of flexible vertex format flags that describes the vertex format of the vertices in the queried mesh.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::GetDeclaration

ID3DXBaseMesh::GetIndexBuffer

Retrieves the data in an index buffer.

```
HRESULT GetIndexBuffer(
  LPDIRECT3DINDEXBUFFER8* ppIB
);
```

Parameters

ppIB
[out, retval] Address of a pointer to an **IDirect3DIndexBuffer8** interface, representing the index buffer object associated with the mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXBaseMesh::GetNumFaces

Retrieves the number of faces in the mesh.

```
DWORD GetNumFaces();
```

Parameters

None.

Return Values

Returns the number of faces in the mesh.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXBaseMesh::GetNumVertices

Retrieves the number of vertices in the mesh.

```
DWORD GetNumVertices();
```

Parameters

None.

Return Values

Returns the number of vertices in the mesh.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXBaseMesh::GetOptions

Retrieves the mesh options enabled for this mesh at creation time.

```
DWORD GetOptions();
```

Parameters

None.

Return Values

Returns a combination of one or more of the following flags, indicating the options enabled for this mesh at creation time.

D3DXMESH_32BIT
: The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXBaseMesh::GetVertexBuffer

Retrieves the data in a vertex buffer.

```
HRESULT GetVertexBuffer(
  LPDIRECT3DVERTEXBUFFER8* ppVB
);
```

Parameters

ppVB
[out, retval] Address of a pointer to an **IDirect3DVertexBuffer8** interface, representing the vertex buffer object associated with the mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXBaseMesh::LockIndexBuffer

Locks an index buffer and obtains a pointer to the index buffer memory.

```
HRESULT LockIndexBuffer(
  DWORD Flags,
```

BYTE** *ppData*
);

Parameters

Flags

[in] A combination of one or more locking flags, describing how the index buffer memory should be locked.

D3DLOCK_DISCARD

The application overwrites, with a write-only operation, every location within the region being locked. This enables resources stored in non-native formats to save the decompression step.

D3DLOCK_NOOVERWRITE

Indicates that no indices that were referred to in drawing calls since the start of the frame or the last lock without this flag will be modified during the lock. This can enable optimizations when the application is only appending data to the index buffer.

D3DLOCK_NOSYSLOCK

The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY

The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

ppData

[out, retval] Address of a pointer to an array of **BYTE** values, filled with the returned index data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Remarks

When working with index buffers, you are allowed to make multiple lock calls. However, you must ensure that the number of lock calls match the number of unlock calls. DrawPrimitive calls will not succeed with any outstanding lock count on any currently set index buffer.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::UnlockIndexBuffer

ID3DXBaseMesh::LockVertexBuffer

Locks a vertex buffer and obtains a pointer to the vertex buffer memory.

```
HRESULT LockVertexBuffer(
  DWORD Flags,
  BYTE** ppData
);
```

Parameters

Flags

[in] A combination of one or more locking flags, indicating how the vertex buffer memory should be locked.

D3DLOCK_DISCARD

The application overwrites, with a write-only operation, every location within the region being locked. This enables resources stored in non-native formats to save the decompression step.

D3DLOCK_NOOVERWRITE

Indicates that no vertices that were referred to in drawing calls since the start of the frame or the last lock without this flag will be modified during the lock. This can enable optimizations when the application is only appending data to the vertex buffer.

D3DLOCK_NOSYSLOCK

The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY
The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

ppData
[out, retval] Address of a pointer to an array of **BYTE** values, filled with the returned vertex data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Remarks

When working with vertex buffers, you are allowed to make multiple lock calls; however, you must ensure that the number of lock calls match the number of unlock calls. DrawPrimitive calls will not succeed with any outstanding lock count on any currently set vertex buffer.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::UnlockVertexBuffer

ID3DXBaseMesh::UnlockIndexBuffer

Unlocks an index buffer.

```
HRESULT UnlockIndexBuffer();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::LockIndexBuffer

ID3DXBaseMesh::UnlockVertexBuffer

Unlocks a vertex buffer.

```
HRESULT UnlockVertexBuffer();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**

- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::LockVertexBuffer

ID3DXBuffer

The **ID3DXBuffer** interface is used as a data buffer, storing vertex, adjacency, and material information during mesh optimization and loading operations. The buffer object is used to return arbitrary length data.

Also, buffer objects are used to return object code and error messages in methods that assemble vertex and pixel shaders.

The **ID3DXBuffer** interface is obtained by calling the **D3DXCreateBuffer** function.

The methods of the **ID3DXBuffer** interface can be organized into the following group.

Information	**GetBufferPointer**
	GetBufferSize

The **ID3DXBuffer** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPD3DXBUFFER** type is defined as a pointer to the **ID3DXBuffer** interface.

```
typedef struct ID3DXBuffer *LPD3DXBUFFER;
```

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXBuffer::GetBufferPointer

Retrieves a pointer to the data in the buffer.

LPVOID GetBufferPointer();

Parameters

None.

Return Values

Returns a pointer to the data in the buffer.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXBuffer::GetBufferSize

Retrieves the total size of the data in the buffer.

DWORD GetBufferSize();

Parameters

None.

Return Values

Returns the total size of the data in the buffer, in bytes.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXEffect

The **ID3DXEffect** interface is used to set and query effects, and to choose techniques. An effect object can contain multiple techniques to render the same effect.

The **ID3DXEffect** interface is obtained by calling **D3DXCreateEffect**

The methods of the **ID3DXEffect** interface can be organized into the following groups.

Copying	**CloneEffect**
Effects	**SetDword**

	SetFloat
	SetMatrix
	SetPixelShader
	SetTexture
	SetVector
	SetVertexShader
Effects Information	**GetDword**
	GetFloat
	GetMatrix
	GetParameterDesc
	GetPixelShader
	GetTexture
	GetVector
	GetVertexShader
Information	**GetDesc**
	GetDevice
Technique	**GetTechnique**
	GetTechniqueDesc

The **ID3DXEffect** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPD3DXEFFECT** type is defined as a pointer to the **ID3DXEffect** interface.

```
typedef struct ID3DXEffect *LPD3DXEFFECT;
```

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateEffect, **D3DXCompileEffect**, **D3DXCompileEffectFromFileA**, **D3DXCompileEffectFromFileW**

ID3DXEffect::CloneEffect

Creates a clone of an effect.

```
STDMETHODIMP CloneEffect(
  LPDIRECT3DDEVICE8 pDevice,
  DWORD             Usage,
  LPD3DXEFFECT*     ppEffect
)
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device associated with the effect.

Usage
[in] Usage control for this effect. The following flag can be set.

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the effect will be rendered using software processing.

ppEffect
[out, retval] Pointer to a **ID3DXEffect** interface, containing the cloned effect.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

ID3DXEffect::GetDesc

Retrieves a description of the effect.

```
HRESULT GetDesc(
  D3DXEFFECT_DESC* pDesc
```

)

Parameters

pDesc
[in, retval] Pointer to a **D3DXEFFECT_DESC** structure, containing the returned description.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC

ID3DXEffect::GetDevice

Retrieves the device associated with the effect.

HRESULT GetDevice(
IDirect3DDevice8** *ppDevice*
)

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface, representing the device associated with the effect.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count for the **IDirect3DDevice8** interface. Be sure to call **IUnknown::Release** when you are done using this **IDirect3DDevice8** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

ID3DXEffect::GetDword

Retrieves a **DWORD** value associated with an effect.

```
HRESULT GetDword(
  DWORD  Name,
  DWORD* pdw
)
```

Parameters

Name
[in] Name of the parameter to retrieve the **DWORD** value from. This parameter must be a four-character code. See Remarks.

pdw
[out, retval] Pointer to a **DWORD**, containing the returned **DWORD**.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXEffect::SetDword

ID3DXEffect::GetFloat

Retrieves a floating-point value associated with an effect.

```
HRESULT GetFloat(
  UINT  Name,
```

```
  FLOAT* pf
);
```

Parameters

Name
[in] Name of the parameter to retrieve the floating-point value from. This parameter must be a four-character code. See Remarks.

pf
[out, retval] Pointer to a **FLOAT**, containing the returned floating-point value.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::SetFloat

ID3DXEffect::GetMatrix

Retrieves a matrix associated with an effect.

```
HRESULT GetMatrix(
  UINT       Name,
  D3DMATRIX* pMat
);
```

Parameters

Name
[in] Name of the parameter to retrieve the matrix from. This parameter must be a four-character code. See Remarks.

pMatrix

[out, retval] Pointer to a **D3DMATRIX** structure, containing the returned matrix.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::SetMatrix

ID3DXEffect::GetParameterDesc

Retrieves information about an particular effect parameter.

```
HRESULT GetParameterDesc(
  UINT              Index,
  D3DXPARAMETER_DESC* pDesc
)
```

Parameters

Index
[in] Index of the parameter to retrieve information about.

pDesc
[out, retval] Pointer to a **D3DXPARAMETER_DESC** structure, specifying information about the parameter.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

ID3DXEffect::GetPixelShader

Retrieves a pixel shader associated with an effect.

```
HRESULT GetPixelShader(
  DWORD  Name,
  DWORD* pHandle
)
```

Parameters

Name
[in] Name of the parameter to retrieve the pixel shader from. This parameter must be a four-character code. See Remarks.

pHandle
[out, retval] Pointer to a **DWORD**, representing the returned pixel shader.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXEffect::SetPixelShader

ID3DXEffect::GetTechnique

Retrieves a technique associated with an effect.

```
HRESULT GetTechnique(
  UINT          Name,
  ID3DXTechnique* ppTechnique
```

```
);
```

Parameters

Name
[in] Name of the technique to retrieve. This parameter must be a four-character code. See Remarks.

ppTechnique
[out, retval] Pointer to an **ID3DXTechnique** interface, containing the returned technique.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXTECHNIQUE_DESC, ID3DXEffect::GetTechniqueDesc

ID3DXEffect::GetTechniqueDesc

Retrieves a technique associated with an effect.

```
HRESULT GetTechniqueDesc(
  UINT              Index,
  D3DXTECHNIQUE_DESC* pDesc
)
```

Parameters

Index
[in] Name of the technique description to retrieve.

pDesc
[out, retval] Pointer to a **D3DXTECHNIQUE_DESC**, containing the technique description.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXTECHNIQUE_DESC, ID3DXEffect::GetTechnique

ID3DXEffect::GetTexture

Retrieves a texture associated with an effect.

```
HRESULT GetTexture(
  UINT                Name,
  IDirect3DBaseTexture8* pTex
);
```

Parameters

Name
[in] Name of the parameter to retrieve the texture from. This parameter must be a four-character code. See Remarks.

pTex
[out, retval] Pointer to an **IDirect3DBaseTexture8** interface, containing the returned texture.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Note

Calling this method will increase the internal reference count on the **IDirect3DBaseTexture8** interface. Be sure to call **IUnknown::Release** when

you are done using this **IDirect3DTexture8** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXEffect::SetTexture

ID3DXEffect::GetVector

Retrieves a vector associated with an effect.

```
HRESULT GetVector(
  UINT        Name,
  D3DXVECTOR4* pVec
);
```

Parameters

Name
[in] Name of the parameter to retrieve the vector from. This parameter must be a four-character code. See Remarks.

pVec
[out, retval] Pointer to a **D3DXVECTOR4** structure, containing the returned vector.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::GetVector

ID3DXEffect::GetVertexShader

Retrieves a vertex shader associated with an effect.

```
HRESULT GetVertexShader(
  DWORD  Name,
  DWORD* pHandle
)
```

Parameters

Name
[in] Name of the parameter to retrieve the vertex shader from. This parameter must be a four-character code. See Remarks.

pHandle
[out, retval] Pointer to a **DWORD**, containing the returned vertex shader.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::SetVertexShader

ID3DXEffect::SetDword

Sets a **DWORD** value for an effect.

```
HRESULT SetDword(
  UINT  Name,
```

```
  DWORD dw
);
```

Parameters

Name

[in] Identifies the name of the effect parameter to set. This parameter must be a four-character code. See Remarks.

dw

[in] Specifies the **DWORD** value to assign to *Name*.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::GetDword

ID3DXEffect::SetFloat

Sets a floating-point value for an effect.

```
HRESULT SetFloat(
  UINT  Name,
  FLOAT f
);
```

Parameters

Name

[in] Identifies the name of the effect parameter to set. This parameter must be a four-character code. See Remarks.

f

[in] Specifies the floating-point value to assign to *Name*.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::GetFloat

ID3DXEffect::SetMatrix

Sets a matrix for an effect.

```
HRESULT SetMatrix(
  UINT Name,
  D3DMATRIX* pMat
);
```

Parameters

Name
[in] Identifies the name of the effect parameter to set. This parameter must be a four-character code. See Remarks.

pMat
[in] Pointer to the **D3DMATRIX** structure to assign to *Name*.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::GetMatrix

ID3DXEffect::SetPixelShader

Sets a pixel shader for an effect.

```
HRESULT SetPixelShader(
  DWORD Name,
  DWORD Handle
)
```

Parameters

Name
[in] Identifies the name of the effect parameter to set. This parameter must be a four-character code. See Remarks.

Handle
[in] Specifies the pixel shader to assign to *Name*.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::GetPixelShader

ID3DXEffect::SetTexture

Sets a texture for an effect.

```
HRESULT SetTexture(
  UINT              Name,
  IDirect3DBaseTexture8 pTexture
);
```

Parameters

Name
[in] Identifies the name of the effect parameter to set. This parameter must be a four-character code. See Remarks.

pTexture
[in] Pointer to an **IDirect3DBaseTexture8** interface, specifying the texture to assign to *Name*.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Note

Calling this method will increase the internal reference count on the **IDirect3DBaseTexture8** interface. Be sure to call **IUnknown::Release** when you are done using this **IDirect3BaseTexture8** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::GetTexture

ID3DXEffect::SetVector

Sets a vector for an effect.

```
HRESULT SetVector(
  UINT        Name,
  D3DXVECTOR4* pVec
);
```

Parameters

Name
[in] Identifies the name of the effect parameter to set. This parameter must be a four-character code. See Remarks.

pVector
[in] Pointer to a **D3DXVECTOR4** structure, specifying the vector to assign to *Name*.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::SetVector

ID3DXEffect::SetVertexShader

Sets a vertex shader for an effect.

```
HRESULT SetVertexShader(
  DWORD Name,
  DWORD Handle
)
```

Parameters

Name
[in] Identifies the name of the effect parameter to set. This parameter must be a four-character code. See Remarks.

Handle
[in] Handle to the vertex shader to assign to *Name*.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXEFFECT_DESC, ID3DXEffect::SetVertexShader

ID3DXFont

The **ID3DXFont** interface is used to encapsulate the textures and resources needed to render a specific font on a specific device.

The **ID3DXFont** interface is obtained by calling the **D3DXCreateFont** or **D3DXCreateFontIndirect** functions.

The methods of the **ID3DXFont** interface can be organized into the following groups.

Drawing	**Begin**
	DrawTextA
	DrawTextW
	End
Information	**GetDevice**
	GetLogFont

The **ID3DXFont** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPD3DXFONT** type is defined as a pointer to the **ID3DXFont** interface.

```
typedef interface ID3DXFont *LPD3DXFONT;
```

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

ID3DXFont::Begin

Prepares a device for drawing text.

```
HRESULT Begin();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

This method is optional. If a DrawText method is called outside of a **Begin** and **ID3DXFont::End** pair, then Microsoft® Direct3D® will call **Begin** and **End**.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXFont::End

ID3DXFont::DrawTextA

Draws formatted ANSI text on a Microsoft® Direct3D® device.

```
INT DrawTextA(
  LPCSTR pString,
  INT Count,
  LPRECT pRect,
  DWORD Format,
  D3DCOLOR Color
);
```

Parameters

pString

[in] Pointer to the ANSI string to draw. If the *Count* parameter is –1, the string must be null-terminated.

If *Format* includes DT_MODIFYSTRING, the function could add up to four additional characters to this string. The buffer containing the string should be large enough to accommodate these extra characters.

Count

[in] Specifies the number of characters in the string. If *Count* is –1, then the *pString* parameter is assumed to be a pointer to a null-terminated string and **DrawTextA** computes the character count automatically.

pRect

[in] Pointer to a **RECT** structure that contains the rectangle, in logical coordinates, in which the text is to be formatted.

Format

[in] Specifies the method of formatting the text. It can be any combination of the following values.

DT_BOTTOM

Justifies the text to the bottom of the rectangle. This value must be combined with DT_SINGLELINE.

DT_CALCRECT

Determines the width and height of the rectangle. If there are multiple lines of text, **DrawTextA** uses the width of the rectangle pointed to by the *pRect* parameter and extends the base of the rectangle to bound the last line of text. If there is only one line of text, **DrawTextA** modifies the right side of the rectangle so that it bounds the last character in the line. In either case, **DrawTextA** returns the height of the formatted text but does not draw the text.

DT_CENTER

Centers text horizontally in the rectangle.

DT_EDITCONTROL

Duplicates the text-displaying characteristics of a multiline edit control. Specifically, the average character width is calculated in the same manner as for an edit control, and the function does not display a partially visible last line.

DT_END_ELLIPSIS or DT_PATH_ELLIPSIS
Truncates the string without adding ellipses so that the result fits in the specified rectangle. The string is not modified unless the DT_MODIFYSTRING flag is specified.

Specify DT_END_ELLIPSIS to truncate characters at the end of the string, or DT_PATH_ELLIPSIS to truncate characters in the middle of the string. If the string contains backslash (\) characters, DT_PATH_ELLIPSIS preserves as much of the text as possible after the last backslash.

DT_EXPANDTABS
Expands tab characters. The default number of characters per tab is eight. The DT_WORD_ELLIPSIS, DT_PATH_ELLIPSIS, and DT_END_ELLIPSIS values cannot be used with the DT_EXPANDTABS value.

DT_EXTERNALLEADING
Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.

DT_HIDEPREFIX
Windows 2000: Ignores the ampersand (&) prefix character in the text. The letter that follows is not underlined, but other mnemonic-prefix characters are still processed.

Example:

input string: "A&bc&&d"

normal: "Abc&d"

HIDEPREFIX: "Abc&d"

Compare with DT_NOPREFIX and DT_PREFIXONLY.

DT_INTERNAL
Uses the system font to calculate text metrics.

DT_LEFT
Aligns text to the left.

DT_MODIFYSTRING
Modifies the string to match the displayed text. This flag has no effect unless the DT_END_ELLIPSIS or DT_PATH_ELLIPSIS flag is specified.

DT_NOCLIP
Draws without clipping. **DrawTextA** is somewhat faster when DT_NOCLIP is used.

DT_NOFULLWIDTHCHARBREAK
Windows 98, Windows 2000: Prevents a line break at a DBCS (double-wide character string), so that the line breaking rule is equivalent to SBCS strings. For example, this can be used in Korean windows for more readability of icon labels. This is effective only if DT_WORDBREAK is specified.

DT_NOPREFIX
Turns off processing of prefix characters. Normally, **DrawTextA** interprets the mnemonic-prefix character & as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off. Compare with DT_HIDEPREFIX and DT_PREFIXONLY.

DT_PREFIXONLY
Windows 2000: Draws only an underline at the position of the character following the ampersand (&) prefix character. Does not draw any character in the string.

Example:

input string: "A&bc&&d"

normal: "Abc&d"

PREFIXONLY: " "

Compare with DT_NOPREFIX and DT_HIDEPREFIX.

DT_RIGHT
Aligns text to the right.

DT_RTLREADING
Displays text in right-to-left reading order for bi-directional text when a Hebrew or Arabic font is selected. The default reading order for all text is left-to-right.

DT_SINGLELINE
Displays text on a single line only. Carriage returns and line feeds do not break the line.

DT_TABSTOP
Sets tab stops. Bits 15–8 (high-order byte of the low-order word) of the *Format* parameter specify the number of characters for each tab. The default number of characters per tab is eight. The DT_CALCRECT, DT_EXTERNALLEADING, DT_INTERNAL, DT_NOCLIP, and DT_NOPREFIX values cannot be used with the DT_TABSTOP value.

DT_TOP
Top-justifies text (single line only).

DT_VCENTER
Centers text vertically (single line only).

DT_WORDBREAK
Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the *pRect* parameter. A carriage return/line feed sequence also breaks the line.

DT_WORD_ELLIPSIS
Truncates text that does not fit in the rectangle and adds ellipses.

Color
[in] **D3DCOLOR** type, specifying the color of the text.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

Remarks

DrawText maps to either **DrawTextA** or **ID3DXFont::DrawTextW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **DrawText** is defined.

```
#ifndef DrawText
#ifdef UNICODE
#define DrawText DrawTextW
#else
#define DrawText DrawTextA
#endif
#endif
```

The **DrawTextA** method uses the device context's selected font, text color, and background color to draw the text. Unless the DT_NOCLIP format is used, **DrawTextA** clips the text so that it does not appear outside the specified rectangle. All formatting is assumed to have multiple lines unless the DT_SINGLELINE format is specified.

If the selected font is too large for the rectangle, the **DrawTextA** method does not attempt to substitute a smaller font.

The **DrawTextA** method supports only fonts whose escapement and orientation are both zero.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXFont::DrawTextW

ID3DXFont::DrawTextW

Draws formatted Unicode™ text on a Microsoft® Direct3D® device.

```
INT DrawTextW(
  LPCWSTR pString,
  INT Count,
  LPRECT pRect,
  DWORD Format,
  D3DCOLOR Color
);
```

Parameters

pString

[in] Pointer to the Unicode string to draw. If the *Count* parameter is –1, the string must be null-terminated.

If *Format* includes DT_MODIFYSTRING, the function could add up to four additional characters to this string. The buffer containing the string should be large enough to accommodate these extra characters.

Count

[in] Specifies the number of characters in the string. If *Count* is –1, then the *pString* parameter is assumed to be a pointer to a null-terminated string and **DrawTextW** computes the character count automatically.

pRect

[in] Pointer to a **RECT** structure that contains the rectangle, in logical coordinates, in which the text is to be formatted.

Format

[in] Specifies the method of formatting the text. It can be any combination of the following values.

DT_BOTTOM

Justifies the text to the bottom of the rectangle. This value must be combined with DT_SINGLELINE.

DT_CALCRECT

Determines the width and height of the rectangle. If there are multiple lines of text, **DrawTextW** uses the width of the rectangle pointed to by the *pRect* parameter and extends the base of the rectangle to bound the last line of text. If there is only one line of text, **DrawTextW** modifies the right side of the rectangle so that it bounds the last character in the line. In either case, **DrawTextW** returns the height of the formatted text but does not draw the text.

DT_CENTER

Centers text horizontally in the rectangle.

DT_EDITCONTROL

Duplicates the text-displaying characteristics of a multiline edit control. Specifically, the average character width is calculated in the same manner as for an edit control, and the function does not display a partially visible last line.

DT_END_ELLIPSIS or DT_PATH_ELLIPSIS
Truncates the string without adding ellipses so that the result fits in the specified rectangle. The string is not modified unless the DT_MODIFYSTRING flag is specified.

Specify DT_END_ELLIPSIS to truncate characters at the end of the string, or DT_PATH_ELLIPSIS to truncate characters in the middle of the string. If the string contains backslash (\) characters, DT_PATH_ELLIPSIS preserves as much of the text as possible after the last backslash.

DT_EXPANDTABS
Expands tab characters. The default number of characters per tab is eight. The DT_WORD_ELLIPSIS, DT_PATH_ELLIPSIS, and DT_END_ELLIPSIS values cannot be used with the DT_EXPANDTABS value.

DT_EXTERNALLEADING
Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.

DT_HIDEPREFIX
Windows 2000: Ignores the ampersand (&) prefix character in the text. The letter that follows is not underlined, but other mnemonic-prefix characters are still processed.

Example:

input string: "A&bc&&d"

normal: "Abc&d"

HIDEPREFIX: "Abc&d"

Compare with DT_NOPREFIX and DT_PREFIXONLY.

DT_INTERNAL
Uses the system font to calculate text metrics.

DT_LEFT
Aligns text to the left.

DT_MODIFYSTRING
Modifies the string to match the displayed text. This flag has no effect unless the DT_END_ELLIPSIS or DT_PATH_ELLIPSIS flag is specified.

DT_NOCLIP
Draws without clipping. **DrawTextW** is somewhat faster when DT_NOCLIP is used.

DT_NOFULLWIDTHCHARBREAK
Windows 98, Windows 2000: Prevents a line break at a DBCS (double-wide character string), so that the line breaking rule is equivalent to SBCS strings. For example, this can be used in Korean windows for more readability of icon labels. This is effective only if DT_WORDBREAK is specified.

DT_NOPREFIX
Turns off processing of prefix characters. Normally, **DrawTextW** interprets the mnemonic-prefix character & as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off. Compare with DT_HIDEPREFIX and DT_PREFIXONLY.

DT_PREFIXONLY
Windows 2000: Draws only an underline at the position of the character following the ampersand (&) prefix character. Does not draw any character in the string.

Example:

input string: "A&bc&&d"

normal: "Abc&d"

PREFIXONLY: " "

Compare with DT_NOPREFIX and DT_HIDEPREFIX.

DT_RIGHT
Aligns text to the right.

DT_RTLREADING
Displays text in right-to-left reading order for bi-directional text when a Hebrew or Arabic font is selected. The default reading order for all text is left-to-right.

DT_SINGLELINE
Displays text on a single line only. Carriage returns and line feeds do not break the line.

DT_TABSTOP
Sets tab stops. Bits 15–8 (high-order byte of the low-order word) of the *Format* parameter specify the number of characters for each tab. The default number of characters per tab is eight. The DT_CALCRECT, DT_EXTERNALLEADING, DT_INTERNAL, DT_NOCLIP, and DT_NOPREFIX values cannot be used with the DT_TABSTOP value.

DT_TOP
Top-justifies text (single line only).

DT_VCENTER
Centers text vertically (single line only).

DT_WORDBREAK
Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the *pRect* parameter. A carriage return/line feed sequence also breaks the line.

DT_WORD_ELLIPSIS
Truncates text that does not fit in the rectangle and adds ellipses.

Color
[in] **D3DCOLOR** type, specifying the color of the text.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

Remarks

DrawText maps to either **DrawTextW** or **ID3DXFont::DrawTextA**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **DrawText** is defined.

```
#ifndef DrawText
#ifdef UNICODE
#define DrawText DrawTextW
#else
#define DrawText DrawTextA
#endif
#endif
```

The **DrawTextW** method uses the device context's selected font, text color, and background color to draw the text. Unless the DT_NOCLIP format is used, **DrawTextW** clips the text so that it does not appear outside the specified rectangle. All formatting is assumed to have multiple lines unless the DT_SINGLELINE format is specified.

If the selected font is too large for the rectangle, the **DrawTextW** method does not attempt to substitute a smaller font.

The **DrawTextW** method supports only fonts whose escapement and orientation are both zero.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXFont::DrawTextA

ID3DXFont::End

Restores the device state to how it was when **ID3DXFont::Begin** was called.

```
HRESULT End();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXFont::Begin

ID3DXFont::GetDevice

Retrieves the Microsoft® Direct3D® device associated with the font object.

```
HRESULT GetDevice(
  LPDIRECT3DDEVICE8* ppDevice
);
```

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface, representing the Direct3D device object associated with the font object.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL can be returned if the parameter is invalid.

Note

Calling this method will increase the internal reference count on the **IDirect3DDevice8** interface. Be sure to call **IUnknown::Release** when you are done using this **IDirect3DDevice8** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

ID3DXFont::GetLogFont

Retrieves the attributes of the font.

```
HRESULT GetLogFont(
  LOGFONT* pLogFont
);
```

Parameters

pLogFont

[in, out] Pointer to the returned **LOGFONT** structure, describing the attributes of the font.

Return Values

If the method succeeds, the return value is D3D_OK.

D3DERR_INVALIDCALL can be returned if the parameter is invalid.

Remarks

For more information on the **LOGFONT** structure, see the Microsoft® Platform Software Development Kit (SDK).

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

ID3DXMatrixStack

Applications use the methods of the **ID3DXMatrixStack** interface to manipulate a matrix stack.

The **ID3DXMatrixStack** interface is obtained by calling the **D3DXCreateMatrixStack** function.

The methods of the **ID3DXMatrixStack** interface can be organized into the following groups.

Matrix Multiplication	**MultMatrix**
	MultMatrixLocal
Miscellaneous	**GetTop**
	LoadIdentity
	LoadMatrix
	Pop
	Push
Rotation	**RotateAxis**
	RotateAxisLocal
	RotateYawPitchRoll
	RotateYawPitchRollLocal
Scaling	**Scale**
	ScaleLocal
Translation	**Translate**
	TranslateLocal

The **ID3DXMatrixStack** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

ID3DXMatrixStack::GetTop

Retrieves the current matrix at the top of the stack.

```
D3DXMATRIX* GetTop();
```

Parameters

None.

Return Values

This method returns a pointer to a **D3DXMATRIX** structure representing the current matrix.

Remarks

The **D3DXMATRIX** pointer returned by this method is not guaranteed to be valid after subsequent stack operations.

Note that this method does not remove the current matrix from the top of the stack; rather, it just returns the current matrix.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::Pop, **ID3DXMatrixStack::Push**

ID3DXMatrixStack::LoadIdentity

Loads identity in the current matrix.

```
HRESULT LoadIdentity();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The identity matrix is a matrix in which all coefficients are 0.0 except the [1,1][2,2] [3,3][4,4] coefficients, which are set to 1.0. The identity matrix is special in that when it is applied to vertices, they are unchanged. The identity matrix is used as the starting point for matrices that will modify vertex values to create rotations, translations, and any other transformations that can be represented by a 4×4 matrix.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::LoadMatrix

ID3DXMatrixStack::LoadMatrix

Loads the given matrix into the current matrix.

```
HRESULT LoadMatrix(
  CONST D3DXMATRIX* pMat
);
```

Parameters

pMat
[in] Pointer to the **D3DXMATRIX** structure loaded into the current matrix.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Note that this method does not add an item to the stack; rather, it replaces the current matrix with the supplied matrix.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::LoadIdentity

ID3DXMatrixStack::MultMatrix

Determines the product of the current matrix and the given matrix.

```
HRESULT MultMatrix(
  CONST D3DXMATRIX* pMat
);
```

Parameters

pMat
[in] Pointer to the **D3DXMATRIX** structure to be multiplied with the current matrix.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method right-multiplies the given matrix to the current matrix (transformation is about the current world origin).

```
m_stack[m_currentPos] = m_stack[m_currentPos] * *pMat;
```

Note that this method does not add an item to the stack; rather, it replaces the current matrix with the product of the current matrix and the given matrix.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::MultMatrixLocal

ID3DXMatrixStack::MultMatrixLocal

Determines the product of the given matrix and the current matrix.

```
HRESULT MultMatrixLocal(
  CONST D3DXMATRIX* pMat
);
```

Parameters

pMat
[in] Pointer to the **D3DXMATRIX** structure to be multiplied with the current matrix.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method left-multiplies the given matrix to the current matrix (transformation is about the local origin of the object).

```
m_stack[m_currentPos] = *pMat * m_stack[m_currentPos];
```

Note that this method does not add an item to the stack; rather, it replaces the current matrix with the product of the given matrix and the current matrix.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::MultMatrix

ID3DXMatrixStack::Pop

Removes the current matrix from the top of the stack.

```
HRESULT Pop();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

Remarks

Note that this method decrements the count of items on the stack by 1, effectively removing the current matrix from the top of the stack and promoting a matrix to the top of the stack. If the current count of items on the stack is 0, then this method returns without performing any action. If the current count of items on the stack is 1, then this method empties the stack.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::GetTop, **ID3DXMatrixStack::Push**

ID3DXMatrixStack::Push

Adds a matrix to the stack.

```
HRESULT Push();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method increments the count of items on the stack by 1, duplicating the current matrix. The stack will grow dynamically as more items are added.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::GetTop, **ID3DXMatrixStack::Pop**

ID3DXMatrixStack::RotateAxis

Determines the product of the current matrix and the computed rotation matrix.

```
HRESULT RotateAxis(
  CONST D3DXVECTOR3* pV,
  FLOAT Angle
);
```

Parameters

pV
[in] Pointer to a **D3DXVECTOR3** structure that identifies the axis angle.

Angle
[in] Angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method right-multiplies the current matrix with the computed rotation matrix, counterclockwise about the given axis with the given angle (rotation is about the current world origin).

```
D3DXMATRIX tmp;

D3DXMatrixRotationAxis( &tmp, pV, angle );
m_stack[m_currentPos] = m_stack[m_currentPos] * tmp;
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::RotateAxisLocal,
ID3DXMatrixStack::RotateYawPitchRoll,
ID3DXMatrixStack::RotateYawPitchRollLocal

ID3DXMatrixStack::RotateAxisLocal

Determines the product of the computed rotation matrix and the current matrix.

```
HRESULT RotateAxisLocal(
  CONST D3DXVECTOR3* pV,
  FLOAT Angle
);
```

Parameters

pV
[in] Pointer to a **D3DXVECTOR3** structure that identifies the axis angle.

Angle
[in] Angle of rotation in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method left-multiplies the current matrix with the computed rotation matrix, counterclockwise about the given axis with the given angle (rotation is about the local origin of the object).

```
D3DXMATRIX tmp;

D3DXMatrixRotationAxis( &tmp, pV, angle );
m_stack[m_currentPos] = tmp * m_stack[m_currentPos];
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::RotateAxis, **ID3DXMatrixStack::RotateYawPitchRoll**, **ID3DXMatrixStack::RotateYawPitchRollLocal**

ID3DXMatrixStack::RotateYawPitchRoll

Determines the product of the current matrix and the computed rotation matrix (composed of a given yaw, pitch, and roll).

```
HRESULT RotateYawPitchRoll(
  FLOAT Yaw,
  FLOAT Pitch,
  FLOAT Roll
);
```

Parameters

Yaw
[in] The yaw around the y-axis in radians.

Pitch
[in] The pitch around the x-axis in radians.

Roll
[in] The roll around the z-axis in radians.

Return Values

If the method succeeds, the return value is D3D_OK.

Remarks

This method right-multiplies the current matrix with the computed rotation matrix. All angles are counterclockwise and rotation is about the current world origin.

```
D3DXMATRIX tmp;

D3DXMatrixRotationYawPitchRoll( &tmp, yaw, pitch, roll );
m_stack[m_currentPos] = m_stack[m_currentPos] * tmp;
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::RotateAxis, **ID3DXMatrixStack::RotateAxisLocal**, **ID3DXMatrixStack::RotateYawPitchRollLocal**

ID3DXMatrixStack::RotateYawPitchRollLocal

Determines the product of the computed rotation matrix (composed of a given yaw, pitch, and roll) and the current matrix.

```
HRESULT RotateYawPitchRollLocal(
  FLOAT Yaw,
  FLOAT Pitch,
  FLOAT Roll
);
```

Parameters

Yaw
[in] The yaw around the y-axis in radians.

Pitch
[in] The pitch around the x-axis in radians.

Roll
[in] The roll around the z-axis in radians.

Return Values

If the method succeeds, the return value is D3D_OK.

Remarks

This method left-multiplies the current matrix with the computed rotation matrix. All angles are counterclockwise and rotation is about the local origin of the object.

```
D3DXMATRIX tmp;

D3DXMatrixRotationYawPitchRoll( &tmp, yaw, pitch, roll );
m_stack[m_currentPos] = tmp * m_stack[m_currentPos];
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::RotateAxis, **ID3DXMatrixStack::RotateAxisLocal**, **ID3DXMatrixStack::RotateYawPitchRoll**

ID3DXMatrixStack::Scale

Determines the product of the current matrix and the computed scale matrix composed from the given point (x, y, and z).

```
HRESULT Scale(
  FLOAT x,
  FLOAT y,
  FLOAT z
);
```

Parameters

x
[in] The scaling component in the x-direction.

y
[in] The scaling component in the y-direction.

z
[in] The scaling component in the z-direction.

Return Values

If the method succeeds, the return value is D3D_OK.

Remarks

This method right-multiplies the current matrix with the computed scale matrix (transformation is about the current world origin).

```
D3DXMATRIX tmp;

D3DXMatrixScaling( &tmp, x, y, z );
m_stack[m_currentPos] = m_stack[m_currentPos] * tmp;
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::ScaleLocal

ID3DXMatrixStack::ScaleLocal

Determines the product of the computed scale matrix composed from the given point (x, y, and z) and the current matrix.

```
HRESULT Scale(
  FLOAT x,
  FLOAT y,
  FLOAT z
);
```

Parameters

x
[in] The scaling component in the x-direction.

y
[in] The scaling component in the y-direction.

z
[in] The scaling component in the z-direction.

Return Values

If the method succeeds, the return value is D3D_OK.

Remarks

This method left-multiplies the current matrix with the computed scale matrix (transformation is about the local origin of the object).

```
D3DXMATRIX tmp;

D3DXMatrixScaling( &tmp, x, y, z );
m_stack[m_currentPos] = tmp * m_stack[m_currentPos];
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::Scale

ID3DXMatrixStack::Translate

Determines the product of the current matrix and the computed translation matrix determined by the given factors (x, y, and z).

```
HRESULT Translate(
  FLOAT x,
  FLOAT y,
  FLOAT z
);
```

Parameters

x
[in] The translation factor in the x-direction.

y
[in] The translation factor in the y-direction.

z
[in] The translation factor in the z-direction.

Return Values

If the method succeeds, the return value is D3D_OK.

Remarks

This method right-multiplies the current matrix with the computed translation matrix (transformation is about the current world origin).

```
D3DXMATRIX tmp;

D3DXMatrixTranslation( &tmp, x, y, z );
m_stack[m_currentPos] = m_stack[m_currentPos] * tmp;
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::TranslateLocal

ID3DXMatrixStack::TranslateLocal

Determines the product of the computed translation matrix determined by the given factors (x, y, and z) and the current matrix.

```
HRESULT TranslateLocal(
  FLOAT x,
  FLOAT y,
  FLOAT z
);
```

Parameters

x
[in] The translation factor in the x-direction.

y
[in] The translation factor in the y-direction.

z
[in] The translation factor in the z-direction.

Return Values

If the method succeeds, the return value is D3D_OK.

Remarks

This method left-multiplies the current matrix with the computed translation matrix (transformation is about the local origin of the object).

```
D3DXMATRIX tmp;

D3DXMatrixTranslation( &tmp, x, y, z );
m_stack[m_currentPos] = tmp * m_stack[m_currentPos];
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMatrixStack::Translate

ID3DXMesh

Applications use the methods of the **ID3DXMesh** interface to manipulate mesh objects.

The **ID3DXMesh** interface is obtained by calling either the **D3DXCreateMesh** or **D3DXCreateMeshFVF** functions.

The **ID3DXMesh** interface inherits the following **ID3DXBaseMesh** methods, which can be organized into the following groups.

Buffers	**GetIndexBuffer**
	GetVertexBuffer
	LockIndexBuffer
	LockVertexBuffer
	UnlockIndexBuffer
	UnlockVertexBuffer
Copying	**CloneMesh**
	CloneMeshFVF
Faces	**GetNumFaces**
Information	**GetDevice**
	GetOptions
Rendering	**DrawSubset**
	GetAttributeTable
Vertices	**GetFVF**
	GetNumVertices

The methods of the **ID3DXMesh** interface can be organized into the following groups.

Locking	**LockAttributeBuffer**
	UnlockAttributeBuffer

Miscellaneous	**ConvertAdjacencyToPointReps**
	ConvertPointRepsToAdjacency
	GenerateAdjacency
Optimization	**Optimize**
	OptimizeInplace

The **ID3DXMesh** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPD3DXMESH** type is defined as a pointer to the **ID3DXMesh** interface.

```
typedef struct ID3DXMesh *LPD3DXMESH;
```

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

Mesh Functions

ID3DXMesh::ConvertAdjacencyToPointReps

Generates point representatives for the mesh.

```
HRESULT ConvertAdjacencyToPointReps(
  CONST DWORD* pAdjacency,
  DWORD* PointRep
);
```

Parameters

pAdjacency
[in] Pointer to the face adjacency array of the mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

PointRep

[in, out] Pointer to a destination buffer for the point representatives of the mesh. The point representatives are stored as an array of indices with one element per vertex.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDMESH

E_OUTOFMEMORY

Remarks

Point representatives are a method of describing mesh adjacency by fusing two vertices with the same x-, y-, and z- coordinates but with different normal coordinates, u- and v-coordinates, color data, and so on. In a perfectly smooth mesh (a mesh without creases, boundaries, or holes), point representatives are redundant. In that case, it is possible to compute adjacent triangles by using a hash table to locate edges that share vertex indices. In the case of creases and texture boundaries, the point representatives are required to give a unique vertex index to use in a hash table for a group of co-located vertices that make up a vertex in the mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMesh::ConvertPointRepsToAdjacency

ID3DXMesh::ConvertPointRepsToAdjacency

Converts point representative data stored in Microsoft® DirectX® (.x) files to face adjacency information that is more flexible for optimization and simplification operations.

```
HRESULT ConvertPointRepsToAdjacency(
  CONST DWORD* pPRep,
  DWORD* pAdjacency
);
```

Parameters

pPRep
[in] Pointer to the point representatives for the mesh. The point representatives are stored as an array of indices with one element per vertex.

pAdjacency
[in, out] Pointer to a destination buffer for the face adjacency array of the mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

Point representatives are a method of describing mesh adjacency by fusing two vertices with the same x-, y-, and z- coordinates but with different normal coordinates, u- and v-coordinates, color data, an so on. In a perfectly smooth mesh (a mesh without creases, boundaries, or holes), point representatives are redundant. In that case, it is possible to compute adjacent triangles by using a hash table to locate edges that share vertex indices. In the case of creases and texture boundaries, the point representatives are required to give a unique vertex index to use in a hash table for a group of co-located vertices that make up a vertex in the mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMesh::ConvertAdjacencyToPointReps

ID3DXMesh::GenerateAdjacency

Uses an epsilon value to generate face adjacency information that is more flexible for optimization and simplification operations.

HRESULT GenerateAdjacency(
FLOAT *Epsilon*,

DWORD* *pAdjacency*
);

Parameters

Epsilon
[in] Separation distance under which vertices are welded. This parameter is currently ignored and uses an epsilon of 0.0.

pAdjacency
[in, out] Pointer to a destination buffer for the face adjacency array of the mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

This method only logically welds the vertices and no changes are made to the mesh. **D3DXWeldVertices** should be called after this method if modifying the mesh is desired.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXMesh::LockAttributeBuffer

Locks an attribute buffer and obtains a pointer to the attribute buffer memory.

HRESULT LockAttributeBuffer(
DWORD *Flags*,
DWORD** *ppData*
);

Parameters

Flags

[in] A combination of one or more locking flags, indicating how the attribute buffer memory should be locked.

D3DLOCK_DISCARD

The application overwrites, with a write-only operation, every location within the region being locked. This enables resources stored in non-native formats to save the decompression step.

D3DLOCK_NOOVERWRITE

Indicates that no information referred to in drawing calls since the start of the frame or the last lock without this flag will be modified during the lock. This can enable optimizations when the application is only appending data to the vertex buffer.

D3DLOCK_NOSYSLOCK

The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but it can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY

The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

ppData

[out, retval] Address of a pointer to an array of **DWORD** values, filled with the returned attribute buffer data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMesh::UnlockAttributeBuffer, **D3DXATTRIBUTERANGE**

ID3DXMesh::Optimize

Controls the reordering of mesh faces and vertices to optimize performance, generating an output mesh.

```
HRESULT Optimize(
  DWORD Flags,
  CONST DWORD* pAdjacencyIn,
  DWORD* pAdjacencyOut,
  DWORD* pFaceRemap,
  LPD3DXBUFFER* ppVertexRemap,
  LPD3DXMESH* ppOptMesh
);
```

Parameters

Flags

[in] A combination of one or more flags, specifying the type of optimization to perform. The following flags are defined.

D3DXMESHOPT_ATTRSORT
Reorders faces to optimize for fewer attribute bundle state changes and enhanced **ID3DXBaseMesh::DrawSubset** performance.

D3DXMESHOPT_COMPACT
Reorders faces to remove unused vertices and faces.

D3DXMESHOPT_IGNOREVERTS
Optimize the faces only; do not optimize the vertices.

D3DXMESHOPT_SHAREVB
Share vertex buffers.

D3DXMESHOPT_STRIPREORDER
Reorders faces to maximize length of adjacent triangles.

D3DXMESHOPT_VERTEXCACHE
Reorders faces to increase the cache hit rate of vertex caches.

Note that the D3DXMESHOPT_STRIPREORDER and D3DXMESHOPT_VERTEXCACHE optimization flags are mutually exclusive.

pAdjacencyIn

[in, out] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the source mesh.

pAdjacencyOut

[in, out] Pointer to a destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

pFaceRemap

[in, out] Pointer to a destination buffer containing the new index for each face.

ppVertexRemap
[out] Address of a pointer to an **ID3DXBuffer** interface; containing the new index for each vertex.

ppOptMesh
[out] Address of a pointer to an **ID3DXMesh** interface, representing the optimized mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

This method is very similar to the **ID3DXBaseMesh::CloneMesh** method, except that it can perform optimization while generating the new clone of the mesh.

The output mesh inherits all of the creation parameters of the input mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXMesh::OptimizeInplace

Controls the reordering of mesh faces and vertices to optimize performance.

```
HRESULT OptimizeInplace(
  DWORD Flags,
  CONST DWORD* pAdjacencyIn,
  DWORD* pAdjacencyOut,
  DWORD* pFaceRemap,
  LPD3DXBUFFER* ppVertexRemap
);
```

Parameters

Flags
[in] A combination of one or more flags, specifying the type of optimization to perform. The following flags are defined.

D3DXMESHOPT_ATTRSORT

Reorders faces to optimize for fewer attribute bundle state changes and enhanced **ID3DXBaseMesh::DrawSubset** performance.

D3DXMESHOPT_COMPACT
Reorders faces to remove unused vertices and faces.

D3DXMESHOPT_IGNOREVERTS
Optimize the faces only; do not optimize the vertices.

D3DXMESHOPT_STRIPREORDER
Reorders faces to maximize length of adjacent triangles.

D3DXMESHOPT_VERTEXCACHE
Reorders faces to increase the cache hit rate of vertex caches.

Note that the D3DXMESHOPT_STRIPREORDER and D3DXMESHOPT_VERTEXCACHE optimization flags are mutually exclusive.

pAdjacencyIn
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the source mesh.

pAdjacencyOut
[out] Pointer to a destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

pFaceRemap
[out] Pointer to a destination buffer containing the new index for each face.

ppVertexRemap
[out] Address of a pointer to an **ID3DXBuffer** interface; containing the new index for each vertex.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_CANNOTATTRSORT

E_OUTOFMEMORY

Notes

This method will fail if the mesh is sharing its vertex buffer with another mesh, unless the D3DXMESHOPT_IGNOREVERTS flag is set in the *Flags* parameter.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXMesh::UnlockAttributeBuffer

Unlocks an attribute buffer.

```
HRESULT UnlockAttributeBuffer();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXMesh::LockAttributeBuffer

ID3DXPMesh

Applications use the methods of the **ID3DXPMesh** interface to manipulate progressive mesh objects. A progressive mesh enables progressive refinement of the mesh object.

The **ID3DXPMesh** interface is obtained by calling the **D3DXGeneratePMesh** function.

The **ID3DXPMesh** interface inherits the following **ID3DXBaseMesh** methods, which can be organized into the following groups.

Buffers	**GetIndexBuffer**
	GetVertexBuffer
	LockIndexBuffer
	LockVertexBuffer
	UnlockIndexBuffer
	UnlockVertexBuffer
Copying	**CloneMesh**
	CloneMeshFVF

Faces	**GetNumFaces**
Information	**GetDevice**
	GetOptions
Rendering	**DrawSubset**
	GetAttributeTable
Vertices	**GetFVF**
	GetNumVertices

The methods of the **ID3DXPMesh** interface can be organized into the following groups.

Copying	**ClonePMesh**
	ClonePMeshFVF
Faces	**GetMaxFaces**
	GetMinFaces
	SetNumFaces
Miscellaneous	**GetAdjacency**
	Save
Optimization	**Optimize**
Vertices	**GetMaxVertices**
	GetMinVertices
	SetNumVertices

The **ID3DXPMesh** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPD3DXPMESH** type is defined as a pointer to the **ID3DXPMesh** interface.

```
typedef struct ID3DXPMesh *LPD3DXPMESH;
```

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

Mesh Functions

ID3DXPMesh::ClonePMesh

Clones a progressive mesh using a declarator.

```
HRESULT ClonePMesh(
  DWORD Options,
  CONST DWORD* pDeclaration,
  LPDIRECT3DDEVICE8 pDevice,
  LPD3DXPMESH* ppCloneMesh
);
```

Parameters

Options

[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers. These meshes will have separate index buffers into the shared vertex buffer.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pDeclaration
[in] Pointer to an array of **DWORD** values, representing the declarator that describes the vertex format of the vertices in the output mesh. This parameter must map directly to an FVF.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device object associated with the mesh.

ppCloneMesh
[out, retval] Address of a pointer to an **ID3DXPMesh** interface, representing the cloned progressive mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::GetDeclaration, **D3DXDeclaratorFromFVF**

ID3DXPMesh::ClonePMeshFVF

Clones a progressive mesh using a flexible vertex format (FVF) code.

```
HRESULT ClonePMeshFVF(
  DWORD Options,
  DWORD FVF,
  LPDIRECT3DDEVICE8 pDevice,
  LPD3DXPMESH* ppCloneMesh
);
```

Parameters

Options
[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS

Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers. These meshes will have separate index buffers into the shared vertex buffer.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

FVF
[in] Combination of flexible vertex format flags that specifies the vertex format for the vertices in the output mesh.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device object associated with the mesh.

ppCloneMesh
[out, retval] Address of a pointer to an **ID3DXPMesh** interface, representing the cloned progressive mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

ClonePMeshFVF can be used to convert a progressive mesh from one FVF to another.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXBaseMesh::GetFVF, **D3DXFVFFromDeclarator**

ID3DXPMesh::GetAdjacency

Returns the face adjacency array of the mesh.

```
HRESULT GetAdjacency(
  DWORD* pAdjacency
);
```

Parameters

pAdjacency
[out] Pointer to the returned face adjacency array of the mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh. The size of the adjacency array is the maximum number of faces multiplied by 3.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXPMesh::GetMaxFaces

ID3DXPMesh::GetMaxFaces

Retrieves the maximum number of faces that the progressive mesh supports.

```
DWORD GetMaxFaces();
```

Parameters

None.

Return Values

Returns the maximum number of faces supported by the progressive mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXPMesh::GetMaxVertices

Retrieves the maximum number of vertices that the progressive mesh supports.

```
DWORD GetMaxVertices();
```

Parameters

None.

Return Values

Returns the maximum number of vertices supported by the progressive mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXPMesh::GetMinFaces

Retrieves the minimum number of faces that the progressive mesh supports.

```
DWORD GetMinFaces();
```

Parameters

None.

Return Values

Returns the minimum number of faces supported by the progressive mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXPMesh::GetMinVertices

Retrieves the minimum number of vertices that the progressive mesh supports.

```
DWORD GetMinVertices();
```

Parameters

None.

Return Values

Returns the minimum number of vertices supported by the progressive mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXPMesh::Optimize

Controls the reordering of mesh faces and vertices to optimize performance, generating an output mesh.

```
HRESULT Optimize(
  DWORD Flags,
  DWORD* pAdjacencyOut,
  DWORD* pFaceRemap,
  LPD3DXBUFFER* ppVertexRemap,
  LPD3DXMESH* ppOptMesh
);
```

Parameters

Flags

[in] A combination of one or more flags, specifying the type of optimization to perform. The following flags are defined.

D3DXMESHOPT_ATTRSORT
Reorders faces to optimize for fewer attribute bundle state changes and enhanced **ID3DXBaseMesh::DrawSubset** performance.

D3DXMESHOPT_COMPACT
Reorders faces to remove unused vertices and faces.

D3DXMESHOPT_IGNOREVERTS
Optimize the faces only; do not optimize the vertices.

D3DXMESHOPT_SHAREVB
Share vertex buffers.

D3DXMESHOPT_STRIPREORDER
Reorders faces to maximize length of adjacent triangles.

D3DXMESHOPT_VERTEXCACHE
Reorders faces to increase the cache hit rate of vertex caches.

Note that the D3DXMESHOPT_STRIPREORDER and D3DXMESHOPT_VERTEXCACHE optimization flags are mutually exclusive.

pAdjacencyOut

[out] Pointer to a destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh. The size of this array is the maximum number of faces multiplied by 3.

pFaceRemap

[out] Pointer to a destination buffer containing the new index for each face.

ppVertexRemap

[out] Address of a pointer to an **ID3DXBuffer** interface; containing the new index for each vertex.

ppOptMesh

[out] Address of a pointer to an **ID3DXMesh** interface, representing the optimized mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXPMesh::Save

Saves the progressive mesh to the specified stream object.

```
HRESULT Save(
  IStream* pStream,
  LPD3DXMATERIAL pMaterials,
  DWORD NumMaterials
);
```

Parameters

pStream
[in] Pointer to an **IStream** interface, representing the stream object to which the file data is written.

pMaterials
[in] Pointer to an array of **D3DXMATERIAL** structures, containing material information to be saved in the stream object.

NumMaterials
[in] Number of **D3DXMATERIAL** structures in the *pMaterials array*

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

For more information on the **IStream** interface, see the Microsoft® Platform Software Development Kit (SDK).

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXPMesh::SetNumFaces

Sets the current level of detail to as close to the specified number of faces as possible.

```
HRESULT SetNumFaces(
  DWORD Faces
);
```

Parameters

Faces
[in] Target number of faces. This value specifies the desired change in the level of detail (LOD).

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

If the number of faces is greater than the maximum number of faces, it is capped at the maximum number of faces returned by **ID3DXPMesh::GetMaxFaces**. If the number of faces is less than the minimum number of faces, it is capped at the minimum number of faces returned by **ID3DXPMesh::GetMinFaces**.

The number of faces after this call may be off by one because some edge collapse may introduce or remove one face or two. For example, if you try setting the number of faces to an intermediate value such as 5, when 4 and 6 are possible, 4 will always be the result.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXPMesh::SetNumVertices

Sets the current level of detail to as close to the specified number of vertices as possible.

```
HRESULT SetNumVertices(
  DWORD Vertices
);
```

Parameters

Vertices
[in] Target number of vertices. This value specifies the desired change in the level of detail (LOD).

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

If the number of vertices is greater than the maximum number of vertices, it is capped at the maximum number of vertices returned by **ID3DXPMesh::GetMaxVertices**. If the number of vertices is less than the minimum number of vertices, it is capped at the minimum number of vertices returned by **ID3DXPMesh::GetMinVertices**.

The number of vertices after this call may be off by one because some edge collapse may introduce or remove one face or two. For example, if you try setting the number of faces to an intermediate value such as 5, when 4 and 6 are possible, 4 will always be the result.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXRenderToSurface

The **ID3DXRenderToSurface** interface is used to generalize the process of rendering to surfaces. If the surface is not a render target, a compatible render target is used, and the result is copied to the surface at the end of the scene.

The **ID3DXRenderToSurface** interface is obtained by calling the **D3DXCreateRenderToSurface** function.

The methods of the **ID3DXRenderToSurface** interface can be organized into the following groups.

Information	**GetDesc**
	GetDevice
Rendering	**BeginScene**
	EndScene

The **ID3DXRenderToSurface** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPD3DXRENDERTOSURFACE** type is defined as a pointer to the **ID3DXRenderToSurface** interface.

```
typedef interface ID3DXRenderToSurface* LPD3DXRENDERTOSURFACE;
```

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

ID3DXRenderToSurface::BeginScene

Begins a scene.

```
HRESULT BeginScene(
  LPDIRECT3DSURFACE8 pSurface,
  CONST D3DVIEWPORT8* pViewport
);
```

Parameters

pSurface
[in] Pointer to an **IDirect3DSurface8** interface, representing the render surface.

pViewport
[in] Pointer to a **D3DVIEWPORT8** structure, describing the viewport for the scene.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXRenderToSurface::EndScene

ID3DXRenderToSurface::EndScene

Ends a scene.

```
HRESULT EndScene();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXRenderToSurface::BeginScene

ID3DXRenderToSurface::GetDesc

Retrieves the parameters of the render surface.

```
HRESULT GetDesc(
  D3DXRTE_DESC* pParameters
);
```

Parameters

pParameters
[out] Pointer to a **D3DXRTS_DESC** structure, describing the parameters of the render surface.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

ID3DXRenderToSurface::GetDevice

Retrieves the Microsoft® Direct3D® device associated with the render surface.

```
HRESULT GetDevice(
  LPDIRECT3DDEVICE8* ppDevice
);
```

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface, representing the Direct3D device object associated with the render surface.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count on the **IDirect3DDevice8** interface. Be sure to call **IUnknown::Release** when you are done using this **IDirect3DDevice8** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh

Applications use the methods of the **ID3DXSkinMesh** interface to manipulate skin mesh objects.

The **ID3DXSkinMesh** interface is obtained by calling one of the following functions.

- **D3DXCreateSkinMesh**
- **D3DXCreateSkinMeshFromMesh**
- **D3DXCreateSkinMeshFVF**

To load a skin mesh from a Microsoft® DirectX® (.x) file data object, use the **D3DXLoadSkinMeshFromXof** function.

Skinning is the technique of having different vertices of a mesh be transformed by multiple transform matrices and the results of the vertex being transformed with each individual matrix blended together to obtain the final transformed vertex.

This process is commonly called skinning because it is used to deform a mesh defining a skin of some character based on the animation of a skeleton of bones. In the simplest case of rigid animation, different parts of the character mesh are associated with a unique skeleton bone and vertices are transformed using the transform of that bone.

When discussing skinning, it is useful to define the following terms.

- *Skin*. The triangle mesh being rendered using deformations. Typically this is the model of the character in some convenient pose.
- *Bone*. A transform matrix that affects one or more vertices of the skin.
- *Skeleton*. A hierarchy of all the bones that affect a skin mesh.
- *Pose*. A set of bone transforms that completely defines the transforms for all the bones that affect a given skin mesh.
- *Initial Pose*. This is the pose in which the mesh was associated with the skeleton. When the skeleton is in this pose, the deformed mesh is identical to the original undeformed mesh.
- *Bone Space Transform*. This transform transforms the mesh into the local space of a particular bone.
- *Bone Weight*. The amount of influence a bone has on a given vertex. A weight of 1 means that the vertex is only affected by that bone and 0 means it is unaffected by the bone. The sum of weights of all the bones that affect a vertex should add up to 1 for all vertices.

A skinned character is defined by a set of meshes and a set of bones that affect the vertices of the meshes. The bones are represented as a transform hierarchy. For each mesh, there is a matrix for every bone that affects it that transforms the mesh into the local coordinate space of the bone. This matrix is the bone space transform of the bone for the mesh. This is defined at the time the skeleton is associated with the mesh in the authoring process.

The methods of the **ID3DXSkinMesh** interface can be organized into the following groups.

Buffers	**GetIndexBuffer**
	GetVertexBuffer
Conversion	**ConvertToBlendedMesh**
	ConvertToIndexedBlendedMesh
Information	**GetBoneInfluence**
	GetDevice
	GetMaxFaceInfluences
	GetMaxVertexInfluences
	GetNumBoneInfluences
	GetNumBones
	GetOptions
Faces	**GetNumFaces**
Locking	**LockAttributeBuffer**
	LockIndexBuffer
	LockVertexBuffer
	UnlockAttributeBuffer
	UnlockIndexBuffer
	UnlockVertexBuffer
Miscellaneous	**GenerateSkinnedMesh**
	GetOriginalMesh
	SetBoneInfluence
	UpdateSkinnedMesh
Vertices	**GetDeclaration**
	GetFVF
	GetNumVertices

The **ID3DXSkinMesh** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPD3DXSKINMESH** type is defined as a pointer to the **ID3DXSkinMesh** interface.

```
typedef struct ID3DXSkinMesh* LPD3DXSKINMESH;
```

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::ConvertToBlendedMesh

Returns a blended mesh.

```
HRESULT ConvertToBlendedMesh(
  DWORD Options,
  CONST LPDWORD pAdjacencyIn,
  LPDWORD pAdjacencyOut,
  DWORD* pNumBoneCombinations,
  LPD3DXBUFFER* ppBoneCombinationTable,
  LPD3DXMESH* ppMesh
);
```

Parameters

Options
[in] A combination of one or more flags specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh

object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_USEHWONLY
Use hardware processing only. This flag should only be specified for a hardware processing device. On a mixed mode device this flag will cause the system to either use hardware only, or if the hardware is not capable it will approximate using the software capabilities.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pAdjacencyIn
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the source mesh.

pAdjacencyOut
[in, out] Pointer to a destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

pNumBoneCombinations

[in, out] Number of entries.

ppBoneCombinationTable
[in, out] Address of a pointer to an array of **D3DXBONECOMBINATION** structures. The count of structures is *pNumBoneCombinations*. Each bone combination table defines the four bones that you can draw at a time.

ppMesh
[in, out] Address of a pointer to an **ID3DXMesh** interface, representing the blended mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be E_OUTOFMEMORY.

Remarks

This method takes a skin mesh and converts it into a blended mesh (a mesh with bone influences) using the Microsoft® Direct3D® vertex blending functionality available in Microsoft® DirectX® 7.*x*. For more information, see Geometry Blending.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::ConvertToIndexedBlendedMesh

Returns an indexed blended mesh.

```
HRESULT ConvertToIndexedBlendedMesh(
  DWORD Options,
  CONST LPDWORD pAdjacencyIn,
  DWORD PaletteSize,
  LPDWORD pAdjacencyOut,
  DWORD* pNumBoneCombinations
  LPD3DXBUFFER* ppBoneCombinationTable
  LPD3DXMESH* ppMesh,
);
```

Parameters

Options
[in] A combination of one or more flags specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pAdjacencyIn
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the source mesh.

PaletteSize
[in] The palette size.

pAdjacencyOut
[in, out] Pointer to a destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

pNumBoneCombinations
[in, out] Number of entries.

ppBoneCombinationTable
[in, out] Address of a pointer to an array of **D3DXBONECOMBINATION** structures. The count of structures is *pNumBoneCombinations*. Each bone combination table defines the four bones that you can draw at a time.

ppMesh
[in, out] Address of a pointer to an **ID3DXMesh** interface, representing the indexed blended mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method takes a skin mesh and converts it into an indexed blended mesh using the Microsoft® Direct3D® indexed vertex blending functionality available in Microsoft DirectX® 8.0. Indexed vertex blending uses indices to index into a matrix palette. For more information, see Geometry Blending. Converting a skin mesh into an indexed blended mesh enables you to render the mesh in a single drawing call.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GenerateSkinnedMesh

Generates a skinned mesh.

```
HRESULT GenerateSkinnedMesh(
  DWORD Options,
  FLOAT minWeight,
  CONST LPDWORD pAdjacencyIn,
```

```
  LPDWORD pAdjacencyOut,
  LPD3DXMESH* ppMesh
);
```

Parameters

Options

[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

minWeight
[in] Sets the minimum weight clamping value.

pAdjacencyIn
[in, out] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the source mesh.

pAdjacencyOut
[in, out] Pointer to a destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

ppMesh
[out, retval] Pointer to an **ID3DXMesh** interface, representing the generated skinned mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The mesh generated by this method is the one that should be supplied to **ID3DXSkinMesh::UpdateSkinnedMesh** each time the skeleton transforms are modified.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::UpdateSkinnedMesh

ID3DXSkinMesh::GetBoneInfluence

Returns a list of values indicating how a bone affects the skin mesh.

```
HRESULT GetBoneInfluence(
  DWORD Bone,
  DWORD* pVertices,
  FLOAT* pWeights
);
```

Parameters

Bone
[in] Bone to query.

pVertices
[in, out] Reference to the list of vertices affected by the bone.

pWeights
[in, out] List of weights associated with each vertex in the list of vertices affected by the bone.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::GetBoneInfluence

ID3DXSkinMesh::GetDeclaration

Retrieves a declaration describing the vertices in the mesh.

```
HRESULT GetDeclaration(
  DWORD Declaration[MAX_FVF_DECL_SIZE]
);
```

Parameters

Declaration

[out] A returned array describing the vertex format of the vertices in the queried mesh. The upper limit of this declarator array is MAX_FVF_DECL_SIZE, limiting the declarator to a maximum of 15 **DWORD**s.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::GetFVF

ID3DXSkinMesh::GetDevice

Retrieves the device associated with the skin mesh.

```
HRESULT GetDevice(
  LPDIRECT3DDEVICE8* ppDevice
```

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface, representing the Microsoft® Direct3D® device object associated with the skin mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in d3dx8mesh.h.
Import Library: Use d3dx8.lib.

ID3DXSkinMesh::GetFVF

Retrieves the flexible vertex format of the vertices in the skin mesh.

```
DWORD GetFVF();
```

Parameters

None.

Return Values

Returns a combination of flexible vertex format flags that describe the vertex format of the vertices in the skin mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::GetDeclaration

ID3DXSkinMesh::GetIndexBuffer

Retrieves the data in an index buffer.

```
HRESULT GetIndexBuffer(
  LPDIRECT3DINDEXBUFFER8* ppIB
);
```

Parameters

ppIB
[out, retval] Address of a pointer to an **IDirect3DIndexBuffer8** interface, representing the index buffer object associated with the skin mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GetMaxFaceInfluences

Return the maximum number of face influences.

```
HRESULT GetMaxFaceInfluences(
  DWORD* maxFaceInfluences
);
```

Parameters

maxFaceInfluences
[out, retval] Maximum number of influences affecting any single face in the skin mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Each face can be affected by *n* bones. The following diagram shows a face affected by 6 bones (the union of bones effecting each vertex of the face).

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GetMaxVertexInfluences

Returns the maximum number of vertex influences.

```
HRESULT GetMaxVertexInfluences(
  DWORD* maxVertexInfluences
);
```

Parameters

maxVertexInfluences
[out, retval] Maximum number of influences affecting any single vertex in the skin mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

Each vertex can be affected by *n* bones. If this method returns a value of 17, up to 17 bones affect a single vertex of the skin mesh. So, there are 17 separate weights on that vertex which add up to 1.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GetNumBoneInfluences

Returns the number of influences (weights) on a given bone.

```
DWORD GetNumBoneInfluences(
  DWORD Bone
);
```

Parameters

Bone
[in] The bone to query.

Return Values

Returns the number of weights on the given bone. If *Bone* is an invalid bone number, **GetNumBoneInfluences** returns 0.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GetNumBones

Returns the number of bones in the skin mesh.

```
DWORD GetNumBones();
```

Parameters

None.

Return Values

Number of bones in the skin mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GetNumFaces

Retrieves the number of faces in the skin mesh.

```
DWORD GetNumFaces();
```

Parameters

None.

Return Values

Returns the number of faces in the skin mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GetNumVertices

Retrieves the number of vertices in the skin mesh.

```
DWORD GetNumVertices();
```

Parameters

None.

Return Values

Returns the number of vertices in the skin mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GetOptions

Retrieves the mesh options enabled for this skin mesh at creation time.

```
DWORD GetOptions();
```

Parameters

None.

Return Values

Returns a combination of one or more of the following flags, indicating the options enabled for this mesh at creation time.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_USEHWONLY
Use hardware processing only. This flag should only be specified for a hardware processing device. On a mixed mode device this flag will cause the system to either use hardware only, or if the hardware is not capable it will approximate using the software capabilities.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GetOriginalMesh

Returns the skin mesh in its original (default) pose.

```
HRESULT GetOriginalMesh(
  LPD3DXMESH* ppMesh
);
```

Parameters

ppMesh
[out, retval] Pointer to an **ID3DXMesh** interface that will contain the original mesh after the method returns. Note that when this method returns, the reference count for the returned mesh is incremented.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count on the **ID3DXMesh** interface. Be sure to call **IUnknown::Release** when you are done using this **ID3DXMesh** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::GetVertexBuffer

Retrieves the data in a vertex buffer.

```
HRESULT GetVertexBuffer(
  LPDIRECT3DVERTEXBUFFER8* ppVB
);
```

Parameters

ppVB

[out, retval] Address of a pointer to an **IDirect3DVertexBuffer8** interface, representing the vertex buffer object associated with the skin mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::LockAttributeBuffer

Locks an attribute buffer and obtains a pointer to the attribute buffer memory.

```
HRESULT LockAttributeBuffer(
  DWORD Flags,
  DWORD** ppData
```

```
);
```

Parameters

Flags

[in] A combination of one or more locking flags, indicating how the attribute buffer memory should be locked.

D3DLOCK_DISCARD

The application overwrites, with a write-only operation, every location within the region being locked. This enables resources stored in non-native formats to save the decompression step.

D3DLOCK_NOOVERWRITE

Indicates that no information referred to in drawing calls since the start of the frame or the last lock without this flag will be modified during the lock. This can enable optimizations when the application is only appending data to the vertex buffer.

D3DLOCK_NOSYSLOCK

The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but it can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY

The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

ppData

[out, retval] Address of a pointer to an array of **DWORD** values, filled with the returned attribute buffer data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::UnlockAttributeBuffer

ID3DXSkinMesh::LockIndexBuffer

Locks an index buffer and obtains a pointer to the index buffer memory.

```
HRESULT LockIndexBuffer(
  DWORD Flags,
  BYTE** ppData
);
```

Parameters

Flags

[in] A combination of one or more locking flags, describing how the index buffer memory should be locked.

D3DLOCK_DISCARD

The application overwrites, with a write-only operation, every location within the region being locked. This enables resources stored in non-native formats to save the decompression step.

D3DLOCK_NOOVERWRITE

Indicates that no indices that were referred to in drawing calls since the start of the frame or the last lock without this flag will be modified during the lock. This can enable optimizations when the application is only appending data to the index buffer.

D3DLOCK_NOSYSLOCK

The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY

The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

ppData

[out, retval] Address of a pointer to an array of **BYTE** values, filled with the returned index data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

When working with index buffers, you are allowed to make multiple lock calls; however, you must ensure that the number of lock calls match the number of unlock calls. DrawPrimitive calls will not succeed with any outstanding lock count on any currently set index buffer.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::UnlockIndexBuffer

ID3DXSkinMesh::LockVertexBuffer

Locks a vertex buffer and obtains a pointer to the vertex buffer memory.

```
HRESULT LockVertexBuffer(
  DWORD Flags,
  BYTE** ppData
);
```

Parameters

Flags

[in] A combination of one or more locking flags, indicating how the vertex buffer memory should be locked.

D3DLOCK_DISCARD

The application overwrites, with a write-only operation, every location within the region being locked. This enables resources stored in non-native formats to save the decompression step.

D3DLOCK_NOOVERWRITE

Indicates that no vertices that were referred to in drawing calls since the start of the frame or the last lock without this flag will be modified during the lock. This can enable optimizations when the application is only appending data to the vertex buffer.

D3DLOCK_NOSYSLOCK

The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for

long duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system responsiveness.

D3DLOCK_READONLY
The application will not write to the buffer. This enables resources stored in non-native formats to save the recompression step when unlocking.

ppData
[out, retval] Address of a pointer to an array of **BYTE** values, filled with the returned vertex data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

When working with vertex buffers, you are allowed to make multiple lock calls; however, you must ensure that the number of lock calls match the number of unlock calls. DrawPrimitive calls will not succeed with any outstanding lock count on any currently set vertex buffer.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::UnlockVertexBuffer

ID3DXSkinMesh::SetBoneInfluence

Sets the bone influence for a bone in the skin mesh.

```
HRESULT SetBoneInfluence(
  DWORD Bone,
  DWORD numInfluences,
  CONST DWORD* pVertices,
  CONST FLOAT* pWeights
);
```

Parameters

Bone
[in] Bone for which to set the influences.

numInfluences

[in] Number of influences to set for the bone.

pVertices
[in] Pointer to the list of vertices used to set the influences (weights).

pWeights
[in] Pointer to the list of weights to set for the provided vertices. These values are used to programmatically fill in the skinning information for the skin mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be E_OUTOFMEMORY.

Remarks

Using this method provides a mechanism to indicate how the given bone affects the given vertices. This method should be called for each bone that affects the mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSkinMesh::UnlockAttributeBuffer

Unlocks an attribute buffer.

```
HRESULT UnlockAttributeBuffer();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::LockAttributeBuffer

ID3DXSkinMesh::UnlockIndexBuffer

Unlocks an index buffer.

```
HRESULT UnlockIndexBuffer();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::LockIndexBuffer

ID3DXSkinMesh::UnlockVertexBuffer

Unlocks a vertex buffer.

```
HRESULT UnlockVertexBuffer();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::LockVertexBuffer

ID3DXSkinMesh::UpdateSkinnedMesh

Updates a skin mesh.

```
HRESULT UpdateSkinnedMesh(
  CONST D3DXMATRIX* pBoneTransforms,
  LPD3DXMESH pMesh
);
```

Parameters

pBoneTransforms
[in] Pointer to an array of **D3DXMATRIX** structures, representing the new bone transforms. The length of this array is the count of all the bones in the skin mesh.

pMesh
[out, retval] Pointer to an **ID3DXMesh** interface, representing the updated skinned mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method updates the vertices blended with the new bone transforms.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::GenerateSkinnedMesh

ID3DXSPMesh

Applications use the methods of the **ID3DXSPMesh** interface to manipulate simplification mesh objects. A simplification mesh is used to simplify a given mesh to a lower number of faces.

The **ID3DXSPMesh** interface is obtained by calling the **D3DXCreateSPMesh** function.

The methods of the **ID3DXSPMesh** interface can be organized into the following groups.

Copying	**CloneMesh**
	CloneMeshFVF
	ClonePMesh
	ClonePMeshFVF
Faces	**GetMaxFaces**
	GetNumFaces
	ReduceFaces
Information	**GetDevice**
	GetOptions
Vertices	**GetDeclaration**
	GetFVF
	GetMaxVertices
	GetNumVertices
	ReduceVertices

The **ID3DXSPMesh** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPD3DXSPMESH** type is defined as a pointer to the **ID3DXSPMesh** interface.

```
typedef struct ID3DXSPMesh *LPD3DXSPMESH;
```

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

Mesh Functions

ID3DXSPMesh::CloneMesh

Clones a mesh using a declarator.

```
HRESULT CloneMesh(
  DWORD Options,
  CONST DWORD* pDeclaration,
  LPDIRECT3DDEVICE8 pD3DDevice,
  DWORD* pAdjacency,
  DWORD* pVertexRemap,
  LPD3DXMESH* ppCloneMesh
);
```

Parameters

Options

[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM

Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers. These meshes will have separate index buffers into the shared vertex buffer.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pDeclaration
[in] Pointer to an array of **DWORD** values, representing the declarator to describe the vertex format of the vertices in the output mesh. This parameter must map directly to an FVF.

pD3DDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device object associated with the mesh.

pAdjacency
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the source mesh.

pVertexRemap
[in] Pointer to an array containing the index for each vertex.

ppCloneMesh
[out, retval] Address of a pointer to an **ID3DXPMesh** interface, representing the cloned mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSPMesh::GetDeclaration, **D3DXDeclaratorFromFVF**

ID3DXSPMesh::CloneMeshFVF

Clones a mesh using a flexible vertex format (FVF) code.

```
HRESULT CloneMeshFVF(
  DWORD Options,
  DWORD FVF,
  LPDIRECT3DDEVICE8 pDevice,
  DWORD* pAdjacency,
  DWORD* pVertexRemap,
  LPD3DXMESH* ppCloneMesh
);
```

Parameters

Options

[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED

Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers. These meshes will have separate index buffers into the shared vertex buffer.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

FVF
[in] Combination of flexible vertex format flags that specifies the vertex format for the vertices in the output mesh.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device object associated with the mesh.

pAdjacency
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the source mesh.

pVertexRemap
[in] Pointer to an array containing the index for each vertex.

ppCloneMesh
[out, retval] Address of a pointer to an **ID3DXPMesh** interface, representing the cloned mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

CloneMeshFVF can be used to convert a mesh from one FVF to another.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSPMesh::GetFVF, **D3DXFVFFromDeclarator**

ID3DXSPMesh::ClonePMesh

Clones a progressive mesh using a declarator.

```
HRESULT ClonePMesh(
  DWORD Options,
  CONST DWORD* pDeclaration,
  LPDIRECT3DDEVICE8 pDevice,
  DWORD* pVertexRemap,
  LPD3DXPMESH* ppCloneMesh
);
```

Parameters

Options
[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers. These meshes will have separate index buffers into the shared vertex buffer.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pDeclaration

[in] Pointer to an array of **DWORD** values, representing the declarator to describe the vertex format of the vertices in the output mesh. This parameter must map directly to an FVF.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device object associated with the mesh.

pVertexRemap
[in] Pointer to an array containing the index for each vertex.

ppCloneMesh
[out, retval] Address of a pointer to an **ID3DXPMesh** interface, representing the cloned progressive mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_CANNOTATTRSORT

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSPMesh::GetDeclaration, **D3DXDeclaratorFromFVF**

ID3DXSPMesh::ClonePMeshFVF

Clones a progressive mesh using a flexible vertex format (FVF) code.

```
HRESULT ClonePMeshFVF(
  DWORD Options,
  DWORD FVF,
  LPDIRECT3DDEVICE8 pDevice,
  DWORD* pVertexRemap,
  LPD3DXPMESH* ppCloneMesh
);
```

Parameters

Options

[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers. These meshes will have separate index buffers into the shared vertex buffer.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

FVF
[in] Combination of flexible vertex format flags that specifies the vertex format for the vertices in the output mesh.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device object associated with the mesh.

pVertexRemap
[in] Pointer to an array containing the index for each vertex.

ppCloneMesh
[out, retval] Address of a pointer to an **ID3DXPMesh** interface, representing the cloned progressive mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_CANNOTATTRSORT

E_OUTOFMEMORY

Remarks

ClonePMeshFVF can be used to convert a progressive mesh from one FVF to another.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSPMesh::GetFVF, **D3DXFVFFromDeclarator**

ID3DXSPMesh::GetDeclaration

Retrieves a declaration describing the vertices in the mesh.

```
HRESULT GetDeclaration(
  DWORD Declaration[MAX_FVF_DECL_SIZE]
);
```

Parameters

Declaration
[out] A returned array describing the vertex format of the vertices in the queried mesh. The upper limit of this declarator array is MAX_FVF_DECL_SIZE, limiting the declarator to a maximum of 15 **DWORD**s.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Applies To

This method applies to the following interfaces, which inherit from **ID3DXBaseMesh**.

- **ID3DXMesh**
- **ID3DXPMesh**

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSPMesh::GetFVF

ID3DXSPMesh::GetDevice

Retrieves the device object associated with the simplification mesh.

```
HRESULT GetDevice(
  LPDIRECT3DDEVICE8* ppDevice
```

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface, representing the Microsoft® Direct3D® device object associated with the simplification mesh.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count on the **IDirect3DDevice8** interface. Be sure to call **IUnknown::Release** when you are done using this **IDirect3DDevice8** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSPMesh::GetFVF

Retrieves the flexible vertex format of the vertices in the simplification mesh.

```
DWORD GetFVF();
```

Parameters

None.

Return Values

Returns a combination of flexible vertex format flags that describe the vertex format of the vertices in the simplification mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSPMesh::GetDeclaration

ID3DXSPMesh::GetMaxFaces

Retrieves the maximum number of faces that the simplification mesh supports.

```
DWORD GetMaxFaces();
```

Parameters

None.

Return Values

Returns the maximum number of faces in the original mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSPMesh::GetMaxVertices

Retrieves the maximum number of vertices that the simplification mesh supports.

```
DWORD GetMaxVertices();
```

Parameters

None.

Return Values

Returns the maximum number of vertices in the original mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSPMesh::GetNumFaces

Retrieves the number of faces in the simplification mesh.

```
DWORD GetNumFaces();
```

Parameters

None.

Return Values

Returns the number of faces in the simplification mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSPMesh::GetNumVertices

Retrieves the number of vertices in the simplification mesh.

```
DWORD GetNumVertices();
```

Parameters

None.

Return Values

Returns the number of vertices in the simplification mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSPMesh::GetOptions

Retrieves the mesh options enabled for this simplification mesh at creation time.

```
DWORD GetOptions();
```

Parameters

None.

Return Values

Returns a combination of one or more of the following flags, indicating the options enabled for this mesh at creation time.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers.

D3DXMESH_VB_SYSTEMMEM

Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSPMesh::ReduceFaces

Reduces the number of faces in a simplification mesh.

```
HRESULT ReduceFaces(
  DWORD Faces
);
```

Parameters

Faces
[in] Number of faces to which to reduce the primitive count.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method performs edge collapses to reduce the number of primitives to the value specified in *Faces*. It is not always possible to reduce a mesh to the requested value due to certain restrictions.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSPMesh::ReduceVertices

Reduces the number of vertices in a simplification mesh.

```
HRESULT ReduceVertices(
```

DWORD *Vertices*
);

Parameters

Vertices
[in] Number of vertices to which to reduce the primitive count.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

This method performs edge collapses to reduce the number of primitives to the value specified in *Vertices*. It is not always possible to reduce a mesh to the requested value due to certain restrictions.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

ID3DXSprite

The **ID3DXSprite** interface provides a set of methods that simplify the process of drawing sprites using Microsoft® Direct3D®.

The **ID3DXSprite** interface is obtained by calling the **D3DXCreateSprite** function.

The methods of the **ID3DXSprite** interface can be organized into the following groups.

Drawing	**Begin**
	Draw
	DrawTransform
	End
Information	**GetDevice**

The **ID3DXSprite** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface

Release

The **LPD3DXSPRITE** type is defined as a pointer to the **ID3DXSprite** interface.

```
typedef interface ID3DXSprite *LPD3DXSPRITE;
```

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

ID3DXSprite::Begin

Prepares a device for drawing sprites.

```
HRESULT Begin();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

Calling **Begin** is optional. If called outside of a Begin/End sequence, the draw functions will internally call **Begin** and **End**. To avoid extra overhead, this method should be used if more than one draw function will be called successively.

ID3DXSprite::Begin cannot be used as a substitute for either IDirect3DDevice8::BeginScene or ID3DXRenderToSurface::BeginScene.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSprite::End

ID3DXSprite::Draw

Draws a simple sprite in screen-space.

```
HRESULT Draw(
  LPDIRECT3DTEXTURE8 pSrcTexture,
  CONST RECT* pSrcRect,
  CONST D3DXVECTOR2* pScaling,
  CONST D3DXVECTOR2* pRotationCenter,
  FLOAT Rotation,
  CONST D3DVECTOR2* pTranslation,
  D3DCOLOR Color
);
```

Parameters

pSrcTexture

[in] Pointer to an **IDirect3DTexture8** interface, representing the source image used for the sprite.

pSrcRect

[in] A pointer to a **RECT** structure that indicates what portion of the source texture to use for the sprite. If this parameter is NULL, then the entire source image is used for the sprite; however, you can specify a sub-rectangle of the source image instead. X and y mirroring can be specified easily by swapping the left and top and the right and bottom parameters of this **RECT** structure.

Before transformation, the size of the sprite is defined by *pSrcRect* with the top-left corner at the origin (0,0).

pScaling

[in] Pointer to a **D3DXVECTOR2** structure, the scaling vector. If this argument is NULL, it is treated as identity.

pRotationCenter

[in] Pointer to a **D3DXVECTOR2** structure, a point that identifies the center of rotation. If this argument is NULL, it is treated as the point (0,0), which is the upper-left corner of the texture.

Rotation

[in] Value that specifies the rotation.

pTranslation

[in] Pointer to a **D3DXVECTOR2** structure, representing the translation. If this argument is NULL, it is treated as identity.

Color

[in] **D3DCOLOR** type. The color and alpha channels are modulated by this value.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

If **ID3DXSprite::Begin** has not been called, this method will internally call **Begin** and **ID3DXSprite::End**. When making successive calls to **ID3DXSprite::Draw** and/or **ID3DXSprite::DrawTransform**, be sure to call **Begin** to avoid the extra overhead of **Draw** and **DrawTransform** internally calling **Begin** and **End** each time.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSprite::DrawTransform

ID3DXSprite::DrawTransform

Provides a mechanism for drawing a sprite that is transformed by a specified matrix.

```
HRESULT DrawTransform(
  LPDIRECT3DTEXTURE8 pSrcTexture,
  CONST RECT* pSrcRect,
  CONST D3DXMATRIX* pTransform,
  D3DCOLOR Color
);
```

Parameters

pSrcTexture
[in] Pointer to an **IDirect3DTexture8** interface, representing the source image used for the sprite.

pSrcRect
[in] A pointer to a **RECT** structure that indicates what portion of the source texture to use for the sprite. If this parameter is NULL, then the entire source image is used for the sprite; however, you can specify a sub-rectangle of the source image instead. X and y mirroring can be specified easily by swapping the left and top and the right and bottom parameters of this **RECT** structure.

Before transformation, the size of the sprite is defined by *pSrcRect* with the top-left corner at the origin (0,0).

pTransform

[in] Pointer to a **D3DXMATRIX** structure, specifying the transformation that will be applied.

Color

[in] **D3DCOLOR** type. The color and alpha channels are modulated by this value.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Remarks

If **ID3DXSprite::Begin** has not been called, this method will internally call **Begin** and **ID3DXSprite::End**. When making successive calls to **ID3DXSprite::Draw** and/or **ID3DXSprite::DrawTransform**, be sure to call **Begin** to avoid the extra overhead of **Draw** and **DrawTransform** internally calling **Begin** and **End** each time.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSprite::Draw

ID3DXSprite::End

Restores the device state to how it was when **ID3DXSprite::Begin** was called.

```
HRESULT End();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

ID3DXSprite::End cannot be used as a substitute for either **IDirect3DDevice8::EndScene** or **ID3DXRenderToSurface::EndScene**.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSprite::Begin

ID3DXSprite::GetDevice

Retrieves the Microsoft® Direct3D® device associated with the sprite object.

```
HRESULT GetDevice(
  LPDIRECT3DDEVICE8* ppDevice
);
```

Parameters

ppDevice
[out, retval] Address of a pointer to an **IDirect3DDevice8** interface, representing the Direct3D device object associated with the sprite object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Note

Calling this method will increase the internal reference count on the **IDirect3DDevice8** interface. Be sure to call **IUnknown::Release** when you are done using this **IDirect3DDevice8** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

ID3DXTechnique

The **ID3DXTechnique** interface is used to query, validate, and apply techniques.

The **ID3DXTechnique** interface is obtained by calling the **ID3DXEffect::GetTechnique** method.

The methods of the **ID3DXTechnique** interface can be organized into the following groups.

Effects	**IsParameterUsed**
Information	**GetDevice**
	GetDesc
	GetPassDesc
Validation	**Validate**
Technique Application	**Begin**
	End
	Pass

The **ID3DXTechnique** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPD3DXTECHNIQUE** type is defined as a pointer to the **ID3DXTechnique** interface.

```
typedef struct ID3DXTechnique* LPD3DXTECHNIQUE;
```

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

ID3DXTechnique::Begin

Begins the application of the technique.

```
HRESULT Begin(
  UINT* pPasses
);
```

Parameters

pPasses

[out, retval] Pointer to a value indicating the number of passes that need to be taken to render the technique.

Return Values

This method always returns the value S_OK.

Remarks

This method returns the number of passes needed to render the technique. The application must incrementally call **ID3DXTechnique::Pass** for each pass before drawing the geometry to which the effect needs to be applied. After all passes are rendered, **ID3DXTechnique::End** must be called.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXTechnique::End, **ID3DXTechnique::Pass**

ID3DXTechnique::End

End the application of the technique.

```
HRESULT End();
```

Parameters

None.

Return Values

This method always returns the value S_OK.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXTechnique::Begin, **ID3DXTechnique::Pass**

ID3DXTechnique::GetDevice

Retrieves the device associated with the technique.

```
HRESULT GetDevice(
  IDirect3DDevice8** ppDevice
);
```

Parameters

ppDevice
Address of a pointer to an **IDirect3DDevice8** interface, representing the device associated with the technique.

Return Values

This method always returns the value S_OK.

Note

Calling this method will increase the internal reference count on the **IDirect3DDevice8** interface. Be sure to call **IUnknown::Release** when you are done using this **IDirect3DDevice8** interface or you will have a memory leak.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

ID3DXTechnique::GetDesc

Retrieves a description of the technique.

```
HRESULT GetDesc(
  D3DXTECHNIQUE_DESC* pDesc
);
```

Parameters

pDesc
[out, retval] A **D3DXTECHNIQUE_DESC** structure, describing the technique.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

ID3DXTechnique::GetPassDesc

Retrieves information about a pass for an effect.

```
HRESULT GetPassDesc(
  UINT          Index,
  D3DXPASS_DESC* pDesc
)
```

Parameters

Index
Index of the pass to retrieve information about.

pDesc
[out, retval] Pointer to a **D3DXPASS_DESC** structure, containing the pass description.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

ID3DXTechnique::IsParameterUsed

Determines if a parameter is used by the technique.

```
BOOL IsParameterUsed(
  DWORD Name
)
```

Parameters

Name
Index of parameter to check. This parameter must be a four-character code. See Remarks.

Return Values

Returns TRUE if parameter is being used and returns FALSE if the parameter is not being used.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

ID3DXTechnique::Pass

Applies the state settings for the specified pass of the technique.

```
HRESULT Pass(
  UINT Index
);
```

Parameters

Pass
[in] Identifies the pass to be applied.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

After calling **Pass**, the application must draw the geometry to the device to which the effect is set.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXTechnique::Begin, **ID3DXTechnique::End**

ID3DXTechnique::Validate

Validates the technique.

```
HRESULT Validate();
```

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

Functions

This section contains reference information for the functions that you need to implement when you work with the Direct3DX utility library. The following functions are implemented.

- Effect Functions
- General Purpose Functions
- Math Functions
- Mesh Functions
- Shader Functions
- Shape-Drawing Functions
- Texturing Functions

Effect Functions

Direct3DX supplies the following effect functions.

- **D3DXCompileEffect**
- **D3DXCompileEffectFromFileA**
- **D3DXCompileEffectFromFileW**
- **D3DXCreateEffect**

D3DXCompileEffect

Compiles an ASCII effect description into a binary form usable by **D3DXCreateEffect**.

```
HRESULT WINAPI D3DXCompileEffect(
  LPCVOID      pSrcData,
  UINT         SrcDataSize,
  LPD3DXBUFFER* ppCompiledEffect,
  LPD3DXBUFFER* ppCompilationErrors
);
```

Parameters

pSrcData
[in] Pointer to source data, representing the effect to compile.

SrcDataSize
[in] Size of the source data, in bytes.

ppCompiledEffect
[out, retval] Address of a pointer to an **ID3DXBuffer** interface, containing the compiled effect.

ppCompilationErrors
[out, retval] Address of a pointer to an **ID3DXBuffer** interface, containing returned ASCII error messages.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

After an effect is compiled, use **D3DXCreateEffect** to obtain a pointer to a **ID3DXEffect** interface.

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCompileEffectFromFileA, D3DXCompileEffectFromFileW, D3DXCreateEffect

D3DXCompileEffectFromFileA

Compiles an effect from a file specified by an ANSI string.

```
HRESULT WINAPI D3DXCompileEffectFromFileA(
  LPCSTR        pSrcFile,
  LPD3DXBUFFER* ppCompiledEffect,
  LPD3DXBUFFER* ppCompilationErrors
);
```

Parameters

pSrcFile
[in] Pointer to an ANSI string that specifies the file from which to create the effect.

ppCompiledEffect
[out, retval] Address of a pointer to an **ID3DXBuffer** interface, containing the compiled effect.

ppCompilationErrors
[out, retval] Address of a pointer to an **ID3DXBuffer** interface, containing the returned ASCII error messages.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

After an effect is compiled, use **D3DXCreateEffect** to obtain a pointer to a **ID3DXEffect** interface.

D3DXCompileEffectFromFile maps to either D3DXCompileEffectFromFileA or D3DXCompileEffectFromFileW, depending on the inclusion or exclusion of the #define UNICODE switch. Include or exclude the #define UNICODE switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how D3DXCompileEffectFromFile is defined.

```
#ifdef UNICODE
#define D3DXCompileEffectFromFile D3DXCompileEffectFromFileW
#else
#define D3DXCompileEffectFromFile D3DXCompileEffectFromFileA
#endif
```

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCompileEffect, D3DXCompileEffectFromFileW, D3DXCreateEffect

D3DXCompileEffectFromFileW

Compiles an effect from a file specified by a Unicode string.

```
HRESULT WINAPI D3DXCompileEffectFromFileW(
  LPCWSTR       pSrcFile,
  LPD3DXBUFFER* ppCompiledEffect,
  LPD3DXBUFFER* ppCompilationErrors
);
```

Parameters

pSrcFile
[in] Pointer to a Unicode string that specifies the file from which to create the effect.

ppCompiledEffect
[out, retval] Address of a pointer to an **ID3DXBuffer** interface, containing the compiled effect.

ppCompilationErrors
[out, retval] Address of a pointer to an **ID3DXBuffer** interface, containing the returned ASCII error messages.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

After an effect is compiled, use **D3DXCreateEffect** to obtain a pointer to a **ID3DXEffect** interface.

D3DXCompileEffectFromFile maps to either D3DXCompileEffectFromFileA or D3DXCompileEffectFromFileW, depending on the inclusion or exclusion of the #define UNICODE switch. Include or exclude the #define UNICODE switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how D3DXCompileEffectFromFile is defined.

```
#ifdef UNICODE
#define D3DXCompileEffectFromFile D3DXCompileEffectFromFileW
#else
#define D3DXCompileEffectFromFile D3DXCompileEffectFromFileA
#endif
```

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCompileEffect, D3DXCompileEffectFromFileA, D3DXCreateEffect

D3DXCreateEffect

Creates an effect using a compiled source data.

```
HRESULT WINAPI D3DXCreateEffect(
  LPDIRECT3DDEVICE8 pDevice,
  LPCVOID           pCompiledEffect,
  UINT              CompiledEffectSize,
  DWORD             Usage,
  LPD3DXEFFECT*     ppEffect
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the effect.

pCompiledEffect
[in] Pointer to the compiled source data from which to create the effect.

CompiledEffectSize
[in] Size of the compiled data, in bytes.

Usage
[in] Usage control for this effect. The following flag can be set:

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the effect will be rendered using software processing.

ppEffect
[out, retval] Address of a pointer to an **ID3DXEffect** interface, representing the created effect.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

Before using **CreateEffect**, an effect must be compiled by using any of the following functions:

- **D3DXCompileEffect**
- **D3DXCompileEffectFromFileA**
- **D3DXCompileEffectFromFileW**

Requirements

Header: Declared in D3dx8effect.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCompileEffect, D3DXCompileEffectFromFileA, D3DXCompileEffectFromFileW

General Purpose Functions

Direct3DX supplies the following general purpose functions.

- **D3DXCreateFont**
- **D3DXCreateFontIndirect**
- **D3DXCreateRenderToSurface**
- **D3DXCreateSprite**
- **D3DXGetErrorStringA**
- **D3DXGetErrorStringW**
- **D3DXGetFVFVertexSize**

D3DXCreateFont

Creates a font object for a device and font.

```
HRESULT D3DXCreateFont(
  LPDIRECT3DDEVICE8 pDevice,
  HFONT hFont,
  LPD3DXFONT* ppFont
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device to be associated with the font object.

hFont
[in] Handle to the font object.

ppFont
[out] Returns a pointer to an **ID3DXFont** interface, representing the created font object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

D3DXCreateFontIndirect

Creates an font object indirectly for a device and font.

```
HRESULT D3DXCreateFontIndirect(
  LPDIRECT3DDEVICE8 pDevice,
  CONST LOGFONT* pLogFont,
  LPD3DXFONT* ppFont
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device to be associated with the font object.

pLogFont
[in] Pointer to a **LOGFONT** structure, describing the attributes of the font object to create.

ppFont
[out] Returns a pointer to an **ID3DXFont** interface, representing the created font object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

For more information on the **LOGFONT** structure, see the Microsoft® Platform Software Development Kit (SDK).

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

D3DXCreateRenderToSurface

Creates a render surface.

```
HRESULT D3DXCreateRenderToSurface(
  LPDIRECT3DDEVICE8 pDevice,
  UINT Width,
  UINT Height,
  D3DFORMAT Format,
  BOOL DepthStencil,
  D3DFORMAT DepthStencilFormat,
  LPD3DXRENDERTOSURFACE* ppRenderToSurface
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device to be associated with the render surface.

Width
[in] Width of the render surface, in pixels.

Height
[in] Height of the render surface, in pixels.

Format
[in] Member of the **D3DFORMAT** enumerated type, describing the pixel format of the render surface.

DepthStencil
[in] If TRUE, the render surface supports a depth-stencil surface. Otherwise this member is set to FALSE.

DepthStencilFormat
[in] If **DepthStencil** is set to TRUE, this parameter is a member of the **D3DFORMAT** enumerated type, describing the depth-stencil format of the render surface.

ppRenderToSurface
[out, retval] Address of a pointer to an **ID3DXRenderToSurface** interface, representing the created render surface.

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

D3DXCreateSprite

Creates a sprite object.

```
HRESULT D3DXCreateSprite(
  LPDIRECT3DDEVICE8 pDevice,
  LPD3DXSPRITE* ppSprite
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device to be associated with the sprite.

ppSprite
[out] Address of a pointer to an **ID3DXSprite** interface, representing the created sprite.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

D3DXGetErrorStringA

Returns the ANSI error string for an HRESULT.

```
HRESULT D3DXGetErrorStringA(
  HRESULT hr,
  LPSTR pBuffer,
  UINT BufferLen
);
```

Parameters

hr
[in] The specified HRESULT error code to decipher.

pBuffer
[out] Pointer to the buffer to fill with the ANSI error string.

BufferLen
[in] Number of characters in the buffer. Any error message longer than this length will be truncated.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be D3DERR_INVALIDCALL.

Remarks

D3DXGetErrorString maps to either **D3DXGetErrorStringA** or **D3DXGetErrorStringW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXGetErrorString** is defined.

```
#ifdef UNICODE
#define D3DXGetErrorString D3DXGetErrorStringW
#else
#define D3DXGetErrorString D3DXGetErrorStringA
#endif
```

This function interprets all Microsoft® Direct3D® HRESULTS.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

D3DXGetErrorStringW

D3DXGetErrorStringW

Returns the Unicode error string for an HRESULT.

```
HRESULT D3DXGetErrorStringW(
  HRESULT hr,
  LPWSTR pBuffer,
```

UINT *BufferLen*
);

Parameters

hr
[in] The specified HRESULT error code to decipher.

pBuffer
[out] Pointer to the buffer to fill in with the Unicode error string.

BufferLen
[in] Number of characters in the buffer. Any error message longer than this length is truncated.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

This function Interprets all Microsoft® Direct3D® HRESULTS.

D3DXGetErrorString maps to either **D3DXGetErrorStringA** or **D3DXGetErrorStringW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXGetErrorString** is defined.

```
#ifdef UNICODE
#define D3DXGetErrorString D3DXGetErrorStringW
#else
#define D3DXGetErrorString D3DXGetErrorStringA
#endif
```

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

D3DXGetErrorStringA

D3DXGetFVFVertexSize

Returns the size of a vertex for a flexible vertex format.

```
UINT D3DXGetFVFVertexSize(
  DWORD FVF
);
```

Parameters

FVF
[in] Flexible vertex format to be queried. A combination of flexible vertex format flags.

Return Values

The flexible vertex format vertex size, in bytes.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

Math Functions

The math library provided by the Direct3DX utility library supplies functions to compute both basic and complicated 3-D mathematical operations.

The 3-D math application functions can be organized into the following groups.

Color	**D3DXColorAdd**
	D3DXColorAdjustContrast
	D3DXColorAdjustSaturation
	D3DXColorLerp
	D3DXColorModulate
	D3DXColorNegative
	D3DXColorScale
	D3DXColorSubtract
Creation	**D3DXCreateMatrixStack**
Plane	**D3DXPlaneDot**
	D3DXPlaneDotCoord
	D3DXPlaneDotNormal
	D3DXPlaneIntersectLine
	D3DXPlaneFromPointNormal

	D3DXPlaneNormalize
	D3DXPlaneFromPoints
	D3DXPlaneTransform
Quaternion	**D3DXQuaternionBaryCentric**
	D3DXQuaternionConjugate
	D3DXQuaternionDot
	D3DXQuaternionExp
	D3DXQuaternionIdentity
	D3DXQuaternionInverse
	D3DXQuaternionIsIdentity
	D3DXQuaternionLength
	D3DXQuaternionLengthSq
	D3DXQuaternionLn
	D3DXQuaternionMultiply
	D3DXQuaternionNormalize
	D3DXQuaternionRotationAxis
	D3DXQuaternionRotationMatrix
	D3DXQuaternionRotationYawPitchRoll
	D3DXQuaternionSlerp
	D3DXQuaternionSquad
	D3DXQuaternionToAxisAngle
2-D Vector	**D3DXVec2Add**
	D3DXVec2BaryCentric
	D3DXVec2CatmullRom
	D3DXVec2CCW
	D3DXVec2Dot
	D3DXVec2Hermite
	D3DXVec2Length
	D3DXVec2LengthSq
	D3DXVec2Lerp
	D3DXVec2Maximize
	D3DXVec2Minimize
	D3DXVec2Normalize
	D3DXVec2Scale
	D3DXVec2Subtract
	D3DXVec2Transform
	D3DXVec2TransformCoord
	D3DXVec2TransformNormal

3-D Vector	**D3DXVec3Add**
	D3DXVec3BaryCentric
	D3DXVec3CatmullRom
	D3DXVec3Cross
	D3DXVec3Dot
	D3DXVec3Hermite
	D3DXVec3Length
	D3DXVec3LengthSq
	D3DXVec3Lerp
	D3DXVec3Maximize
	D3DXVec3Minimize
	D3DXVec3Normalize
	D3DXVec3Project
	D3DXVec3Scale
	D3DXVec3Subtract
	D3DXVec3Transform
	D3DXVec3TransformCoord
	D3DXVec3TransformNormal
	D3DXVec3Unproject
4-D Matrix	**D3DXMatrixAffineTransformation**
	D3DXMatrixfDeterminant
	D3DXMatrixIdentity
	D3DXMatrixInverse
	D3DXMatrixIsIdentity
	D3DXMatrixLookAtRH
	D3DXMatrixLookAtLH
	D3DXMatrixMultiply
	D3DXMatrixOrthoRH
	D3DXMatrixOrthoLH
	D3DXMatrixOrthoOffCenterRH
	D3DXMatrixOrthoOffCenterLH
	D3DXMatrixPerspectiveRH
	D3DXMatrixPerspectiveLH
	D3DXMatrixPerspectiveFovRH
	D3DXMatrixPerspectiveFovLH
	D3DXMatrixPerspectiveOffCenterRH
	D3DXMatrixPerspectiveOffCenterLH
	D3DXMatrixReflect

	D3DXMatrixRotationAxis
	D3DXMatrixRotationQuaternion
	D3DXMatrixRotationX
	D3DXMatrixRotationY
	D3DXMatrixRotationYawPitchRoll
	D3DXMatrixRotationZ
	D3DXMatrixScaling
	D3DXMatrixShadow
	D3DXMatrixTransformation
	D3DXMatrixTranslation
	D3DXMatrixTranspose
4-D Vector	**D3DXVec4Add**
	D3DXVec4BaryCentric
	D3DXVec4CatmullRom
	D3DXVec4Cross
	D3DXVec4Dot
	D3DXVec4Hermite
	D3DXVec4Length
	D3DXVec4LengthSq
	D3DXVec4Lerp
	D3DXVec4Maximize
	D3DXVec4Minimize
	D3DXVec4Normalize
	D3DXVec4Scale
	D3DXVec4Subtract
	D3DXVec4Transform

Note

All the math functions can take the same object as the passed [in] and returned [out] parameters. Also, out parameters are typically returned as return values, so that the output of one math function may be used as a parameter for another math function.

D3DXColorAdd

Adds two color values together to create a new color value.

D3DXCOLOR* D3DXColorAdd(
  D3DXCOLOR* *pOut*,
  CONST D3DXCOLOR* *pC1*,

```
  CONST D3DXCOLOR* pC2
);
```

Parameters

pOut
[in, out] Pointer to a **D3DXCOLOR** structure that is the result of the operation.

pC1
[in] Pointer to a source **D3DXCOLOR** structure.

pC2
[in] Pointer to a source **D3DXCOLOR** structure.

Return Values

This function returns a pointer to a **D3DXCOLOR** structure that is the sum of two color values.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXColorAdd** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXColorModulate, **D3DXColorSubtract**

D3DXColorAdjustContrast

Adjusts the contrast value of a color.

```
D3DXCOLOR* D3DXColorAdjustContrast(
  D3DXCOLOR* pOut,
  D3DXCOLOR* pC,
  FLOAT c
);
```

Parameters

pOut
[in, out] Pointer to a **D3DXCOLOR** structure that is the result of the operation.

pC
[in] Pointer to a source **D3DXCOLOR** structure.

c

[in] Contrast value. This parameter linearly interpolates between 50 percent gray and the color, *pC*. There are not limits on the value of *c*. If this parameter is zero, then the returned color is 50 percent gray. If this parameter is 1, then the returned color is the original color.

Return Values

This function returns a pointer to a **D3DXCOLOR** structure that is the result of the contrast adjustment.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXColorAdjustContrast** function can be used as a parameter for another function.

This function interpolates the red, green, and blue color components of a **D3DXCOLOR** structure between 50 percent gray and a specified contrast value, as shown in the following example.

```
pOut->r = 0.5f + c * (pC->r - 0.5f);
```

If *c* is greater than 0 and less than 1, the contrast is decreased. If *c* is greater than 1, then the contrast is increased.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXColorAdjustSaturation

D3DXColorAdjustSaturation

Adjusts the saturation value of a color.

```
D3DXCOLOR* D3DXColorAdjustSaturation(
  D3DXCOLOR* pOut,
  D3DXCOLOR* pC,
  FLOAT s
);
```

Parameters

pOut

[in, out] Pointer to a **D3DXCOLOR** structure that is the result of the operation.

pC

[in] Pointer to a source **D3DXCOLOR** structure.

s

[in] Saturation value. This parameter linearly interpolates between the color converted to gray-scale and the original color, *pC*. There are no limits on the value of *s*. If *s* is 0, then the returned color is the gray-scale color. If *s* is 1, the returned color is the original color.

Return Values

This function returns a pointer to a **D3DXCOLOR** structure that is the result of the saturation adjustment.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXColorAdjustSaturation** function can be used as a parameter for another function.

This function interpolates the red, green, and blue color components of a **D3DXCOLOR** structure between an unsaturated color and a color, as shown in the following example.

```
// Approximate values for each component's contribution to luminance.
// Based upon the NTSC standard described in ITU-R Recommendation BT.709.
FLOAT grey = pC->r * 0.2125f + pC->g * 0.7154f + pC->b * 0.0721f;


pOut->r = grey + s * (pC->r - grey);
```

If *s* is greater than 0 and less than 1, the saturation is decreased. If *s* is greater than 1, the saturation is increased.

The gray-scale color is computed as: r = g = b = 0.2125*r + 0.7154*g + 0.0721*b.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXColorAdjustContrast

D3DXColorLerp

Uses linear interpolation to create a color value.

D3DXCOLOR* D3DXColorLerp(

```
  D3DXCOLOR* pOut,
  CONST D3DXCOLOR* pC1,
  CONST D3DXCOLOR* pC2,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to a **D3DXCOLOR** structure that is the result of the operation.

pC1
[in] Pointer to a source **D3DXCOLOR** structure.

pC2
[in] Pointer to a source **D3DXCOLOR** structure.

s
[in] Parameter that linearly interpolates between the colors, *pC1* and *pC2*, treating them both as 4-D vectors. There are no limits on the value of *s*.

Return Values

This function returns a pointer to a **D3DXCOLOR** structure that is the result of the linear interpolation.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXColorLerp** function can be used as a parameter for another function.

This function interpolates the red, green, blue, and alpha components of a **D3DXCOLOR** structure between two colors, as shown in the following example.

```
pOut->r = pC1->r + s * (pC2->r - pC1->r);
```

If you are linearly interpolating between the colors A and B, and *s* is 0, the resulting color is A. If *s* is 1, the resulting color is color B.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXColorModulate, **D3DXColorNegative**, **D3DXColorScale**

D3DXColorModulate

Blends two colors.

```
D3DXCOLOR* D3DXColorModulate(
  D3DXCOLOR* pOut,
  CONST D3DXCOLOR* pC1,
  CONST D3DXCOLOR* pC2
);
```

Parameters

pOut
[in, out] Pointer to a **D3DXCOLOR** structure that is the result of the operation.

pC1
[in] Pointer to a source **D3DXCOLOR** structure.

pC2
[in] Pointer to a source **D3DXCOLOR** structure.

Return Values

This function returns a pointer to a **D3DXCOLOR** structure that is the result of the blending operation.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXColorModulate** function can be used as a parameter for another function.

This function blends together two colors by multiplying matching color components, as shown in the following example.

```
pOut->r = pC1->r * pC2->r;
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXColorLerp, **D3DXColorNegative**, **D3DXColorScale**

D3DXColorNegative

Creates the negative color value of a color value.

```
D3DXCOLOR* D3DXColorNegative(
  D3DXCOLOR* pOut,
  CONST D3DXCOLOR* pC
);
```

Parameters

pOut
[in, out] Pointer to a **D3DXCOLOR** structure that is the result of the operation.

pC
[in] Pointer to a source **D3DXCOLOR** structure.

Return Values

This function returns a pointer to a **D3DXCOLOR** structure that is the negative color value of the color value.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXColorNegative** function can be used as a parameter for another function.

This function returns the negative color value by subtracting 1.0 from the color components of the **D3DXCOLOR** structure, as shown in the following example.

```
pOut->r = 1.0f - pC->r;
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXColorLerp, **D3DXColorModulate**, **D3DXColorScale**

D3DXColorScale

Scales a color value.

```
D3DXCOLOR* D3DXColorScale(
  D3DXCOLOR* pOut,
  CONST D3DXCOLOR* pC,
  FLOAT s
);
```

Parameters

pOut

[in, out] Pointer to a **D3DXCOLOR** structure that is the result of the operation.

pC

[in] Pointer to a source **D3DXCOLOR** structure.

s

[in] Scale factor. It scales the color, treating it like a 4-D vector. There are no limits on the value of *s*. If *s* is 1, the resulting color is the original color.

Return Values

This function returns a pointer to a **D3DXCOLOR** structure that is the scaled color value.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXColorScale** function can be used as a parameter for another function.

This function computes the scaled color value by multiplying the color components of the **D3DXCOLOR** structure by the specified scale factor, as shown in the following example.

```
pOut->r = pC->r * s;
```

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXColorLerp, **D3DXColorModulate**, **D3DXColorNegative**

D3DXColorSubtract

Subtracts two color values to create a new color value.

```
D3DXCOLOR* D3DXColorSubtract(
  D3DXCOLOR* pOut,
  CONST D3DXCOLOR* pC1,
  CONST D3DXCOLOR* pC2
);
```

Parameters

pOut

[in, out] Pointer to a **D3DXCOLOR** structure that is the result of the operation.

pC1

[in] Pointer to a source **D3DXCOLOR** structure.

pC2

[in] Pointer to a source **D3DXCOLOR** structure.

Return Values

This function returns a pointer to a **D3DXCOLOR** structure that is the difference between two color values.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXColorSubtract** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXColorAdd

D3DXCreateMatrixStack

Creates an instance of the **ID3DXMatrixStack** interface.

```
HRESULT D3DXCreateMatrixStack(
  DWORD Flags,
  ID3DXMatrixStack** ppStack
);
```

Parameters

Flags

[in] Not implemented. Specify zero.

ppStack

[out] Address of a pointer filled with an **ID3DXMatrixStack** interface pointer if the function succeeds.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXMatrixAffineTransformation

Builds an affine transformation matrix.

```
D3DXMATRIX* D3DXMatrixAffineTransformation(
  D3DXMATRIX* pOut,
  FLOAT scaling,
  CONST D3DXVECTOR3* pRotationCenter,
  CONST D3DXQUATERNION* pRotation,
  CONST D3DXVECTOR3* pTranslation
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

scaling
[in] Scaling factor.

pRotationCenter
[in] Pointer to a **D3DXVECTOR3** structure, a point identifying the center of rotation. If this argument is NULL, it is treated as identity.

pRotation
[in] Pointer to a **D3DXQUATERNION** structure that specifies the rotation. If this argument is NULL, it is treated as identity.

pTranslation
[in] Pointer to a **D3DXVECTOR3** structure, representing the translation. If this argument is NULL, it is treated as identity.

Return Values

Pointer to a **D3DXMATRIX** structure that is an affine transformation matrix.

Remarks

The **D3DXMatrixAffineTransformation** function calculates the affine transformation matrix with the following formula: Ms * Mrc-1 * Mr * Mrc * Mt,

where Ms is the scaling matrix, Mrc is the center of rotation matrix, Mr is the rotation matrix, and Mt is the translation matrix.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixAffineTransformation** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXMatrixfDeterminant

Returns the determinant of a matrix.

```
FLOAT D3DXMatrixfDeterminant(
  CONST D3DXMATRIX* pM
);
```

Parameters

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

The value of the matrix, the determinant.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXMatrixIdentity

Creates an identity matrix.

```
D3DXMATRIX* D3DXMatrixIdentity(
  D3DXMATRIX* pOut
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Return Values

Pointer to a **D3DXMATRIX** structure that is the identity matrix.

Remarks

The identity matrix is a matrix in which all coefficients are 0 except the [1,1][2,2] [3,3][4,4] coefficients, which are set to 1. The identity matrix is special in that when it is applied to vertices, they are unchanged. The identity matrix is used as the starting point for matrices that will modify vertex values to create rotations, translations, and any other transformations that can be represented by a 4×4 matrix.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixIdentity** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixIsIdentity

D3DXMatrixInverse

Calculates the inverse of a matrix.

```
D3DXMATRIX* D3DXMatrixInverse(
  D3DXMATRIX* pOut,
  FLOAT* pDeterminant,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pDeterminant
[in, out] After this function returns, pointer to a **FLOAT** value that is the determinant of the matrix, *pM*.

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXMATRIX** structure that is the inverse of the matrix.

Remarks

This function accepts any arbitrary matrix.

Matrix inversion may fail, in which case NULL is returned by the **D3DXMatrixInverse** function. The determinant of *pM* is returned if the *pDeterminant* parameter is non-NULL.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixInverse** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXMatrixIsIdentity

Determines if a matrix is an identity matrix.

```
BOOL D3DXMatrixIsIdentity(
  CONST D3DXMATRIX* pM
);
```

Parameters

pM
[in] Pointer to the **D3DXMATRIX** structure that will be tested for identity.

Return Values

If the matrix is an identity matrix, this function returns TRUE. Otherwise, this function returns FALSE.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixIdentity

D3DXMatrixLookAtRH

Builds a right-handed, look-at matrix.

```
D3DXMATRIX* D3DXMatrixLookAtRH(
  D3DXMATRIX* pOut,
  CONST D3DXVECTOR3* pEye,
  CONST D3DXVECTOR3* pAt,
  CONST D3DXVECTOR3* pUp
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pEye
[in] Pointer to the **D3DXVECTOR3** structure that defines the eye point. This value is used in translation.

pAt
[in] Pointer to the **D3DXVECTOR3** structure that defines the camera look-at target.

pUp
[in] Pointer to the **D3DXVECTOR3** structure that defines the current world's up, usually [0, 1, 0].

Return Values

Pointer to a **D3DXMATRIX** structure that is a right-handed, look-at matrix.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixLookAtRH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixLookAtLH

D3DXMatrixLookAtLH

Builds a left-handed, look-at matrix.

```
D3DXMATRIX* D3DXMatrixLookAtLH(
  D3DXMATRIX* pOut,
  CONST D3DXVECTOR3* pEye,
  CONST D3DXVECTOR3* pAt,
  CONST D3DXVECTOR3* pUp
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pEye
[in] Pointer to the **D3DXVECTOR3** structure that defines the eye point. This value is used in translation.

pAt
[in] Pointer to the **D3DXVECTOR3** structure that defines the camera look-at target.

pUp
[in] Pointer to the **D3DXVECTOR3** structure that defines the current world's up, usually [0, 1, 0].

Return Values

Pointer to a **D3DXMATRIX** structure that is a left-handed, look-at matrix.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixLookAtLH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixLookAtRH

D3DXMatrixMultiply

Determines the product of two matrices.

```
D3DXMATRIX* D3DXMatrixMultiply(
  D3DXMATRIX* pOut,
  CONST D3DXMATRIX* pM1,
  CONST D3DXMATRIX* pM2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pM1
[in] Pointer to a source **D3DXMATRIX** structure.

pM2
[in] Pointer to a source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXMATRIX** structure that is the product of two matrices.

Remarks

The result represents the transformation M2 followed by the transformation M1 (Out = M1 * M2).

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixMultiply** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionMultiply

D3DXMatrixOrthoRH

Builds a right-handed orthogonal projection matrix.

```
D3DXMATRIX* D3DXMatrixOrthoRH(
  D3DXMATRIX* pOut,
```

```
  FLOAT w,
  FLOAT h,
  FLOAT zn,
  FLOAT zf
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

w
[in] Width of the view-volume.

h
[in] Height of the view-volume.

zn
[in] Minimum z-value of the view volume.

zf
[in] Maximum z-value of the view volume.

Return Values

Pointer to a **D3DXMATRIX** structure that is a right-handed orthogonal projection matrix.

Remarks

An orthogonal matrix is an invertible matrix for which the inverse of the matrix is equal to the transpose of the matrix.

All the parameters of the **D3DXMatrixOrthoRH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixOrthoRH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixOrthoLH, **D3DXMatrixOrthoOffCenterRH**, **D3DXMatrixOrthoOffCenterLH**

D3DXMatrixOrthoLH

Builds a left-handed orthogonal projection matrix.

```
D3DXMATRIX* D3DXMatrixOrthoLH(
  D3DXMATRIX* pOut,
  FLOAT w,
  FLOAT h,
  FLOAT zn,
  FLOAT zf
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

w
[in] Width of the view-volume.

h
[in] Height of the view-volume.

zn
[in] Minimum z-value of the view volume.

zf
[in] Maximum z-value of the view volume.

Return Values

Pointer to a **D3DXMATRIX** structure that is a left-handed orthogonal projection matrix.

Remarks

An orthogonal matrix is an invertible matrix for which the inverse of the matrix is equal to the transpose of the matrix.

All the parameters of the **D3DXMatrixOrthoLH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixOrthoLH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixOrthoRH, **D3DXMatrixOrthoOffCenterRH**, **D3DXMatrixOrthoOffCenterLH**

D3DXMatrixOrthoOffCenterRH

Builds a customized, right-handed orthogonal projection matrix.

```
D3DXMATRIX* D3DXMatrixOrthoOffCenterRH(
  D3DXMATRIX* pOut,
  FLOAT l,
  FLOAT r,
  FLOAT t,
  FLOAT b,
  FLOAT zn,
  FLOAT zf
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

l
[in] Minimum x-value of view-volume.

r
[in] Maximum x-value of view-volume.

t
[in] Minimum y-value of view-volume.

b
[in] Maximum y-value of view-volume.

zn
[in] Minimum z-value of the view volume.

zf
[in] Maximum z-value of the view volume.

Return Values

Pointer to a **D3DXMATRIX** structure that is a customized, right-handed orthogonal projection matrix.

Remarks

An orthogonal matrix is an invertible matrix for which the inverse of the matrix is equal to the transpose of the matrix.

The **D3DXMatrixOrthoRH** function is a special case of the **D3DXMatrixOrthoOffCenterRH** function. To create the same projection using **D3DXMatrixOrthoOffCenterRH**, use the following values: *l* = -w/2, *r* = w/2, *b* = -h/2, and *t* = h/2.

All the parameters of the **D3DXMatrixOrthoOffCenterRH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixOrthoOffCenterRH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixOrthoRH, **D3DXMatrixOrthoLH**, **D3DXMatrixOrthoOffCenterLH**

D3DXMatrixOrthoOffCenterLH

Builds a customized, left-handed orthogonal projection matrix.

```
D3DXMATRIX* D3DXMatrixOrthoOffCenterLH(
  D3DXMATRIX* pOut,
  FLOAT l,
  FLOAT r,
  FLOAT t,
  FLOAT b,
  FLOAT zn,
  FLOAT zf
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

l
[in] Minimum x-value of view-volume.

r
[in] Maximum x-value of view-volume.

t
[in] Minimum y-value of view-volume.

b

[in] Maximum y-value of view-volume.

zn

[in] Minimum z-value of the view volume.

zf

[in] Maximum z-value of the view volume.

Return Values

Pointer to a **D3DXMATRIX** structure that is a customized, left-handed orthogonal projection matrix.

Remarks

An orthogonal matrix is an invertible matrix for which the inverse of the matrix is equal to the transpose of the matrix.

The **D3DXMatrixOrthoLH** function is a special case of the **D3DXMatrixOrthoOffCenterLH** function. To create the same projection using **D3DXMatrixOrthoOffCenterLH**, use the following values: l = -w/2, r = w/2, b = -h/2, and t = h/2.

All the parameters of the **D3DXMatrixOrthoOffCenterLH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixOrthoOffCenterLH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixOrthoRH, **D3DXMatrixOrthoLH**, **D3DXMatrixOrthoOffCenterRH**

D3DXMatrixPerspectiveRH

Builds a right-handed perspective projection matrix.

```
D3DXMATRIX* D3DXMatrixPerspectiveRH(
  D3DXMATRIX* pOut,
  FLOAT w,
  FLOAT h,
  FLOAT zn,
  FLOAT zf
```

);

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

w
[in] Width of the view-volume at the near view-plane.

h
[in] Height of the view-volume at the near view-plane.

zn
[in] Z-value of the near view-plane.

zf
[in] Z-value of the far view-plane.

Return Values

Pointer to a **D3DXMATRIX** structure that is a right-handed perspective projection matrix.

Remarks

All the parameters of the **D3DXMatrixPerspectiveRH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixPerspectiveRH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixPerspectiveLH, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterRH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveFovLH

Builds a left-handed perspective projection matrix based on a field of view (FOV).

D3DXMATRIX* D3DXMatrixPerspectiveFovLH(
D3DXMATRIX* *pOut*,

FLOAT *fovy*,
FLOAT *Aspect*,
FLOAT *zn*,
FLOAT *zf*
);

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

fovy
[in] Field of view, in radians.

Aspect
[in] Aspect ratio.

zn
[in] Z-value of the near view-plane.

zf
[in] Z-value of the far view-plane.

Return Values

Pointer to a **D3DXMATRIX** structure that is a left-handed perspective projection matrix.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixPerspectiveFovLH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveLH**, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveOffCenterRH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveFovRH

Builds a right-handed perspective projection matrix based on a field of view (FOV).

D3DXMATRIX* D3DXMatrixPerspectiveFovRH(

```
  D3DXMATRIX* pOut,
  FLOAT fovy,
  FLOAT Aspect,
  FLOAT zn,
  FLOAT zf
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

fovy
[in] Field of view, in radians.

Aspect
[in] Aspect ratio.

zn
[in] Z-value of the near view-plane.

zf
[in] Z-value of the far view-plane.

Return Values

Pointer to a **D3DXMATRIX** structure that is a right-handed perspective projection matrix.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixPerspectiveFovRH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveLH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterRH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveLH

Builds a left-handed perspective projection matrix

```
D3DXMATRIX* D3DXMatrixPerspectiveLH(
  D3DXMATRIX* pOut,
  FLOAT w,
  FLOAT h,
  FLOAT zn,
  FLOAT zf
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

w
[in] Width of the view-volume at the near view-plane.

h
[in] Height of the view-volume at the near view-plane.

zn
[in] Z-value of the near view-plane.

zf
[in] Z-value of the far view-plane.

Return Values

Pointer to a **D3DXMATRIX** structure that is a left-handed perspective projection matrix.

Remarks

All the parameters of the **D3DXMatrixPerspectiveLH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixPerspectiveLH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterRH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveOffCenterRH

Builds a customized, right-handed perspective projection matrix.

```
D3DXMATRIX* D3DXMatrixPerspectiveOffCenterRH(
  D3DXMATRIX* pOut,
  FLOAT l,
  FLOAT r,
  FLOAT t,
  FLOAT b,
  FLOAT zn,
  FLOAT zf
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

l
[in] X-value of the near view-plane.

r
[in] X-value of the far view-plane.

t
[in] Y-value of the near view-plane.

b
[in] Y-value of the far view-plane.

zn
[in] Z-value of the near view-plane.

zf
[in] Z-value of the far view-plane.

Return Values

Pointer to a **D3DXMATRIX** structure that is a customized, right-handed perspective projection matrix.

Remarks

All the parameters of the **D3DXMatrixPerspectiveOffCenterRH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixPerspectiveOffCenterRH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveLH**, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveOffCenterLH

Builds a customized, left-handed perspective projection matrix.

```
D3DXMATRIX* D3DXMatrixPerspectiveOffCenterLH(
  D3DXMATRIX* pOut,
  FLOAT l,
  FLOAT r,
  FLOAT t,
  FLOAT b,
  FLOAT zn,
  FLOAT zf
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

l
[in] X-value of the near view-plane.

r
[in] X-value of the far view-plane.

t
[in] Y-value of the near view-plane.

b
[in] Y-value of the far view-plane.

zn
[in] Z-value of the near view-plane.

zf
[in] Z-value of the far view-plane.

Return Values

Pointer to a **D3DXMATRIX** structure that is a customized, left-handed perspective projection matrix.

Remarks

All the parameters of the **D3DXMatrixPerspectiveOffCenterLH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixPerspectiveOffCenterLH** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveLH**, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterRH**

D3DXMatrixReflect

Builds a matrix that reflects the coordinate system about a plane.

```
D3DXMATRIX* D3DXMatrixReflect(
  D3DXMATRIX* pOut,
  CONST D3DXPLANE* pPlane
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pPlane
[in] Pointer to the source **D3DXPLANE** structure.

Return Values

Pointer to a **D3DXMATRIX** structure that reflects the coordinate system about the source plane.

Remarks

This function normalizes the plane equation before it creates the reflected matrix.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixReflect** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXMatrixRotationAxis

Builds a matrix that rotates around an arbitrary axis.

```
D3DXMATRIX* D3DXMatrixRotationAxis(
  D3DXMATRIX* pOut,
  CONST D3DXVECTOR3* pV,
  FLOAT Angle
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pV
[in] Pointer to the **D3DXVECTOR3** structure that identifies the axis angle.

Angle
[in] Angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return Values

Pointer to a **D3DXMATRIX** structure rotated around the specified axis.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixRotationAxis** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixRotationQuaternion, **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, **D3DXMatrixRotationYawPitchRoll**, **D3DXMatrixRotationZ**

D3DXMatrixRotationQuaternion

Builds a matrix from a quaternion.

```
D3DXMATRIX* D3DXMatrixRotationQuaternion(
  D3DXMATRIX* pOut,
  CONST D3DXQUATERNION* pQ
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pQ
[in] Pointer to the source **D3DXQUATERNION** structure.

Return Values

Pointer to a **D3DXMATRIX** structure built from the source quaternion.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixRotationQuaternion** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, **D3DXMatrixRotationYawPitchRoll**, **D3DXMatrixRotationZ**

D3DXMatrixRotationX

Builds a matrix that rotates around the x-axis.

```
D3DXMATRIX* D3DXMatrixRotationX(
```

```
  D3DXMATRIX* pOut,
  FLOAT Angle
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Angle
[in] Angle of rotation in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return Values

Pointer to a **D3DXMATRIX** structure rotated around the x-axis.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixRotationX** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationQuaternion**, **D3DXMatrixRotationY**, **D3DXMatrixRotationYawPitchRoll**, **D3DXMatrixRotationZ**

D3DXMatrixRotationY

Builds a matrix that rotates around the y-axis.

```
D3DXMATRIX* D3DXMatrixRotationY(
  D3DXMATRIX* pOut,
  FLOAT Angle
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Angle
[in] Angle of rotation in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return Values

Pointer to a **D3DXMATRIX** structure rotated around the y-axis.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixRotationY** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationQuaternion**, **D3DXMatrixRotationX**, **D3DXMatrixRotationYawPitchRoll**, **D3DXMatrixRotationZ**

D3DXMatrixRotationYawPitchRoll

Builds a matrix with a specified yaw, pitch, and roll.

```
D3DXMATRIX* D3DXMatrixRotationYawPitchRoll(
  D3DXMATRIX* pOut,
  FLOAT Yaw,
  FLOAT Pitch,
  FLOAT Roll,
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Yaw
[in] Yaw around the y-axis, in radians.

Pitch
[in] Pitch around the x-axis, in radians.

Roll
[in] Roll around the z-axis, in radians.

Return Values

Pointer to a **D3DXMATRIX** structure with the specified yaw, pitch, and roll.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixRotationYawPitchRoll** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationQuaternion**, **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, **D3DXMatrixRotationZ**

D3DXMatrixRotationZ

Builds a matrix that rotates around the z-axis.

```
D3DXMATRIX* D3DXMatrixRotationZ(
  D3DXMATRIX* pOut,
  FLOAT Angle
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

Angle
[in] Angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return Values

Pointer to a **D3DXMATRIX** structure rotated around the z-axis.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixRotationZ** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationQuaternion**, **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, **D3DXMatrixRotationYawPitchRoll**

D3DXMatrixScaling

Builds a matrix that scales along the x-, y-, and z-axes.

```
D3DXMATRIX* D3DXMatrixScaling(
  D3DXMATRIX* pOut,
  FLOAT sx,
  FLOAT sy,
  FLOAT sz,
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

sx
[in] Scaling factor that is applied along the x-axis.

sy
[in] Scaling factor that is applied along the y-axis.

sz
[in] Scaling factor that is applied along the z-axis.

Return Values

Pointer to a **D3DXMATRIX** structure that is the scaled matrix.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixScaling** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXMatrixShadow

Builds a matrix that flattens geometry into a plane.

```
D3DXMATRIX* D3DXMatrixShadow(
  D3DXMATRIX* pOut,
  CONST D3DXVECTOR4* pLight,
  CONST D3DXPLANE* pPlane
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pLight
[in] Pointer to a **D3DXVECTOR4** structure describing the light's position.

pPlane
[in] Pointer to the source **D3DXPLANE** structure.

Return Values

Pointer to a **D3DXMATRIX** structure that flattens geometry into a plane.

Remarks

The **D3DXMatrixShadow** function flattens geometry into a plane, as if casting a shadow from a light.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixShadow** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXMatrixTransformation

Builds a transformation matrix.

```
D3DXMATRIX* D3DXMatrixTransformation(
```

```
  D3DXMATRIX* pOut,
  CONST D3DXVECTOR3* pScalingCenter,
  CONST D3DXQUATERNION* pScalingRotation,
  CONST D3DXVECTOR3* pScaling,
  CONST D3DXVECTOR3* pRotationCenter,
  CONST D3DXQUATERNION* pRotation,
  CONST D3DXVECTOR3* pTranslation,
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pScalingCenter
[in] Pointer to a **D3DXVECTOR3** structure, identifying the scaling center point. If this argument is NULL, it is treated as identity.

pScalingRotation
[in] Pointer to a **D3DXQUATERNION** structure that specifies the scaling rotation If this argument is NULL, it is treated as identity.

pScaling
[in] Pointer to a **D3DXVECTOR3** structure, the scaling vector. If this argument is NULL, it is treated as identity.

pRotationCenter
[in] Pointer to a **D3DXVECTOR3** structure, a point that identifies the center of rotation. If this argument is NULL, it is treated as identity.

pRotation
[in] Pointer to a **D3DXQUATERNION** structure that specifies the rotation. If this argument is NULL, it is treated as identity.

pTranslation
[in] Pointer to a **D3DXVECTOR3** structure, representing the translation. If this argument is NULL, it is treated as identity.

Return Values

Pointer to a **D3DXMATRIX** structure that is the transformation matrix.

Remarks

The **D3DXMatrixTransformation** function calculates the transformation matrix with the following formula: Msc-1 * Msr-1 * Ms * Msr * Msc * Mrc-1 * Mr * Mrc * Mt, where Msc is the center scaling matrix, Msr is the scaling rotation matrix, Ms is the scaling matrix, Mrc is the center of rotation matrix, Mr is the rotation matrix, and Mt is the translation matrix.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixTransformation** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixAffineTransformation

D3DXMatrixTranslation

Builds a matrix using the specified offsets.

```
D3DXMATRIX* D3DXMatrixTranslation(
  D3DXMATRIX* pOut,
  FLOAT x,
  FLOAT y,
  FLOAT z,
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

x
[in] X-coordinate offset.

y
[in] Y-coordinate offset.

z
[in] Z-coordinate offset.

Return Values

Pointer to a **D3DXMATRIX** structure that is the translated matrix.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixTranslation** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXMatrixTranspose

Returns the matrix transpose of a matrix.

```
D3DXMATRIX* D3DXMatrixTranspose(
  D3DXMATRIX* pOut,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXMATRIX** structure that is the result of the operation.

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to the **D3DXMATRIX** structure that is the matrix transpose of the matrix.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXMatrixTranspose** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXPlaneDot

Computes the dot-product of a plane and a 4-D vector.

```
FLOAT D3DXPlaneDot(
  CONST D3DXPLANE* pP,
  CONST D3DXVECTOR4* pV
);
```

Parameters

pP

[in] Pointer to a source **D3DXPLANE** structure.

pV

[in] Pointer to a **D3DXVECTOR4** structure.

Return Values

The dot-product of the plane and 4-D vector.

Remarks

Given a plane (a, b, c, d) and a 4-D vector (x, y, z, w) the return value of this function is a*x + b*y + c*z + d*w. The **D3DXPlaneDot** function is useful for determining the plane's relationship with a homogeneous coordinate. For example, this function could be used to determine if a particular coordinate is on a particular plane, or on which side of a particular plane a particular coordinate lies.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXPlaneDotCoord, **D3DXPlaneDotNormal**

D3DXPlaneDotCoord

Computes the dot-product of a plane and a 3-D vector. The *w* parameter of the vector is assumed to be 1.

```
FLOAT D3DXPlaneDotCoord(
  CONST D3DXPLANE* pP,
  CONST D3DXVECTOR3* pV
);
```

Parameters

pP

[in] Pointer to a source **D3DXPLANE** structure.

pV

[in] Pointer to a source **D3DXVECTOR3** structure.

Return Values

The dot-product of the plane and 3-D vector.

Remarks

Given a plane (a, b, c, d) and a 3-D vector (x, y, z) the return value of this function is a*x + b*y + c*z + d*1. The **D3DXPlaneDotCoord** function is useful for determining the plane's relationship with a coordinate in 3-D space.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXPlaneDot, **D3DXPlaneDotNormal**

D3DXPlaneDotNormal

Computes the dot-product of a plane and a 3-D vector. The *w* parameter of the vector is assumed to be 0.

```
FLOAT D3DXPlaneDotNormal(
  CONST D3DXPLANE* pP,
  CONST D3DXVECTOR3* pV
);
```

Parameters

pP
[in] Pointer to a source **D3DXPLANE** structure.

pV
[in] Pointer to a source **D3DXVECTOR3** structure.

Return Values

The dot-product of the plane and 3-D vector.

Remarks

Given a plane (a, b, c, d) and a 3-D vector (x, y, z) the return value of this function is a*x + b*y + c*z + d*0. The **D3DXPlaneDotNormal** function is useful for calculating the angle between the normal of the plane, and another normal.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXPlaneDot, **D3DXPlaneDotCoord**

D3DXPlaneFromPointNormal

Constructs a plane from a point and a normal.

```
D3DXPLANE* D3DXPlaneFromPointNormal(
  D3DXPLANE* pOut,
  CONST D3DXVECTOR3* pPoint,
  CONST D3DXVECTOR3* pNormal
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXPLANE** structure that is the result of the operation.

pPoint
[in] Pointer to a **D3DXVECTOR3** structure, defining the point used to construct the plane.

pNormal
[in] Pointer to a **D3DXVECTOR3** structure, defining the normal used to construct the plane.

Return Values

Pointer to the **D3DXPLANE** structure constructed from the point and the normal.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXPlaneFromPointNormal** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXPlaneFromPoints

D3DXPlaneFromPoints

Constructs a plane from three points.

```
D3DXPLANE* D3DXPlaneFromPoints(
  D3DXPLANE* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2,
  CONST D3DXVECTOR3* pV3
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXPLANE** structure that is the result of the operation.

pV1
[in] Pointer to a **D3DXVECTOR3** structure, defining one of the points used to construct the plane.

pV2
[in] Pointer to a **D3DXVECTOR3** structure, defining one of the points used to construct the plane.

pV3
[in] Pointer to a **D3DXVECTOR3** structure, defining one of the points used to construct the plane.

Return Values

Pointer to the **D3DXPLANE** structure constructed from the given points.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXPlaneFromPoints** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXPlaneFromPointNormal

D3DXPlaneIntersectLine

Finds the intersection between a plane and a line.

```
D3DXVECTOR3* D3DXPlaneIntersectLine(
  D3DXVECTOR3* pOut,
  CONST D3DXPLANE* pP,
```

```
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2
);
```

Parameters

pOut
[in, out] Pointer to a **D3DXVECTOR3** structure, identifying the intersection between the specified plane and line.

pP
[in] Pointer to the source **D3DXPLANE** structure.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure, defining a line starting point.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure, defining a line ending point.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the intersection between the specified plane and line.

Remarks

If the line is parallel to the plane, NULL is returned.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXPlaneIntersectLine** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXPlaneNormalize

Returns the normal of a plane.

```
D3DXPLANE* D3DXPlaneNormalize(
  D3DXPLANE* pOut,
  CONST D3DXPLANE* pP
);
```

Parameters

pOut

[in, out] Pointer to the **D3DXPLANE** structure that is the result of the operation.

pP

[in] Pointer to the source **D3DXPLANE** structure.

Return Values

Pointer to a **D3DXPLANE** structure, representing the normal of the plane.

Remarks

D3DXPlaneNormalize normalizes a plane so that |a,b,c| == 1.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXPlaneNormalize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXPlaneTransform

Transforms a plane by a given matrix.

```
D3DXPLANE* D3DXPlaneTransform(
  D3DXPLANE* pOut,
  CONST D3DXPLANE* pP,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut

[in, out] Pointer to the **D3DXPLANE** structure that is the result of the operation.

pP

[in] Pointer to the source **D3DXPLANE** structure.

pM

[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXPLANE** structure, representing the transformed plane.

Remarks

The matrix, *pM*, must be an affine transform.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXPlaneTransform** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXQuaternionBaryCentric

Returns a quaternion in Barycentric coordinates.

```
D3DXQUATERNION* D3DXQuaternionBaryCentric(
  D3DXQUATERNION* pOut,
  CONST D3DXQUATERNION* pQ1,
  CONST D3DXQUATERNION* pQ2,
  CONST D3DXQUATERNION* pQ3,
  CONST D3DXQUATERNION* pQ4,
  FLOAT f,
  FLOAT g
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pQ1
[in] Pointer to a source **D3DXQUATERNION** structure.

pQ2
[in] Pointer to a source **D3DXQUATERNION** structure.

pQ3
[in] Pointer to a source **D3DXQUATERNION** structure.

pQ4
[in] Pointer to a source **D3DXQUATERNION** structure.

f
[in] Weighting factor. See Remarks.

g
[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXQUATERNION** structure in Barycentric coordinates.

Remarks

To compute the Barycentric coordinates, the **D3DXQuaternionBaryCentric** function implements the following series of spherical linear interpolation operations:
Slerp(Slerp(Q1, Q2, f+g), Slerp(Q1, Q3, f+g), g/(f+g))

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionBaryCentric** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXQuaternionConjugate

Returns the conjugate of a quaternion.

```
D3DXQUATERNION* D3DXQuaternionConjugate(
  D3DXQUATERNION* pOut,
  CONST D3DXQUATERNION* pQ
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pQ
[in] Pointer to the source **D3DXQUATERNION** structure.

Return Values

Pointer to a **D3DXQUATERNION** structure that is the conjugate of the quaternion.

Remarks

Given a quaternion (x, y, z, w), the **D3DXQuaternionConjugate** function will return the quaternion (-x, -y, -z, w).

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionConjugate** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionInverse

D3DXQuaternionDot

Returns the dot-product of two quaternions.

```
FLOAT D3DXQuaternionDot(
  D3DXQUATERNION* pQ1,
  CONST D3DXQUATERNION* pQ2
);
```

Parameters

pQ1
[out] Pointer to a source **D3DXQUATERNION** structure.

pQ2
[in] Pointer to a source **D3DXQUATERNION** structure.

Return Values

The dot-product of two quaternions.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXQuaternionExp

Calculates the exponential.

```
D3DXQUATERNION* D3DXQuaternionExp(
  D3DXQUATERNION* pOut,
  CONST D3DXQUATERNION* pQ
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pQ
[in] Pointer to the source **D3DXQUATERNION** structure.

Return Values

Pointer to a **D3DXQUATERNION** structure that is the exponential.

Remarks

The **D3DXQuaternionExp** function works only with pure quaternions, where *w* == 0.

This function is implemented in the following manner.

```
q = (0, theta * v)
exp(q) = (cos(theta), sin(theta) * v)
```

The **D3DXQuaternionExp** and **D3DXQuaternionLn** functions are useful when using the **D3DXQuaternionSquad** function. Given a set of quaternion keys (q0, q1, q2, ..., q*n*), you can compute the inner quadrangle points (a1, a2, a3, ..., a*n*-1) as shown in the following example, to insure that the tangents are continuous across adjacent segments.

```
      a1   a2   a3
  q0   q1   q2   q3   q4

a[i] = q[i] * exp( -(ln(inv(q[i])*q[i+1]) + ln(inv(q[i])*q[i-1])) / 4 )
```

Once (a1, a2, a3, ...) are computed, you can use the results to interpolate along the curve.

```
qt = Squad(t, q[i], a[i], a[i+1], q[i+1])
```

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionExp** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionLn, **D3DXQuaternionSquad**

D3DXQuaternionIdentity

Returns the identity quaternion.

```
D3DXQUATERNION* D3DXQuaternionIdentity(
  D3DXQUATERNION* pOut
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

Return Values

Pointer to the **D3DXQUATERNION** structure that is the identity quaternion.

Remarks

Given a quaternion (x, y, z, w), the **D3DXQuaternionIdentity** function will return the quaternion (0, 0, 0, 1).

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionIdentity** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionIsIdentity

D3DXQuaternionInverse

Conjugates and renormalizes a quaternion.

```
D3DXQUATERNION* D3DXQuaternionInverse(
  D3DXQUATERNION* pOut,
  CONST D3DXQUATERNION* pQ
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pQ
[in] Pointer to the source **D3DXQUATERNION** structure.

Return Values

Pointer to a **D3DXQUATERNION** structure that is the inverse quaternion of the quaternion.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionInverse** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionConjugate, **D3DXQuaternionNormalize**

D3DXQuaternionIsIdentity

Determines if a quaternion is an identity quaternion.

```
BOOL D3DXQuaternionIsIdentity(
  CONST D3DXQUATERNION* pQ
);
```

Parameters

pQ
[in] Pointer to the **D3DXQUATERNION** structure that will be tested for identity.

Return Values

If the quaternion is an identity quaternion, this function returns TRUE. Otherwise, this function returns FALSE.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionIdentity

D3DXQuaternionLength

Returns the length of a quaternion.

```
FLOAT D3DXQuaternionLength(
  CONST D3DXQUATERNION* pQ
);
```

Parameters

pQ
[in] Pointer to the source **D3DXQUATERNION** structure.

Return Values

The quaternion's length.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionLengthSq

D3DXQuaternionLengthSq

Returns the square of the length of a quaternion.

```
FLOAT D3DXQuaternionLengthSq(
  CONST D3DXQUATERNION* pQ
);
```

Parameters

pQ
[in] Pointer to the source **D3DXQUATERNION** structure.

Return Values

The quaternion's squared length.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionLength

D3DXQuaternionLn

Calculates the natural logarithm.

```
D3DXQUATERNION* D3DXQuaternionLn(
  D3DXQUATERNION* pOut,
  CONST D3DXQUATERNION* pQ
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pQ
[in] Pointer to the source **D3DXQUATERNION** structure.

Return Values

Pointer to a **D3DXQUATERNION** structure that is the natural logarithm.

Remarks

The **D3DXQuaternionLn** function works only for unit quaternions.

This function is implemented in the following manner.

```
q = (cos(theta), sin(theta) * v)
|v| = 1
ln(q) = (0, theta * v)
```

The **D3DXQuaternionExp** and **D3DXQuaternionLn** functions are useful when using the **D3DXQuaternionSquad** function. Given a set of quaternion keys (q0, q1, q2, ..., q*n*), you can compute the inner quadrangle points (a1, a2, a3, ..., a*n*-1) to insure that the tangents are continuous across adjacent segments.

```
      a1   a2   a3
 q0   q1   q2   q3   q4

a[i] = q[i] * exp( -(ln(inv(q[i])*q[i+1]) + ln(inv(q[i])*q[i-1])) / 4 )
```

Once (a1, a2, a3, ...) are computed, you can use the results to interpolate along the curve.

```
qt = Squad(t, q[i], a[i], a[i+1], q[i+1])
```

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionLn** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionExp, **D3DXQuaternionSquad**

D3DXQuaternionMultiply

Multiplies two quaternions.

```
D3DXQUATERNION* D3DXQuaternionMultiply(
  D3DXQUATERNION* pOut,
  CONST D3DXQUATERNION* pQ1,
  CONST D3DXQUATERNION* pQ2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pQ1
[in] Pointer to a source **D3DXQUATERNION** structure.

pQ2
[in] Pointer to a source **D3DXQUATERNION** structure.

Return Values

Pointer to a **D3DXQUATERNION** structure that is the product of two quaternions.

Remarks

The result represents the rotation Q2 followed by the rotation Q1 (Out = Q2 * Q1). The output is actually Q2*Q1 (not Q1*Q2). This is done so that **D3DXQuaternionMultiply** maintain the same semantics as **D3DXMatrixMultiply** because unit quaternions can be considered as another way to represent rotation matrices.

Transformations are concatenated in the same order for both the **D3DXQuaternionMultiply** and **D3DXMatrixMultiply** functions. For example,

assuming mX and mY represent the same rotations as qX and qY, both m and q will represent the same rotations.

```
D3DXMatrixMultiply(&m, &mX, &mY);

D3DXQuaternionMultiply(&q, &qX, &qY);
```

The multiplication of quaternions is not commutative.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionMultiply** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXMatrixMultiply

D3DXQuaternionNormalize

Returns the normal of a quaternion.

```
D3DXQUATERNION* D3DXQuaternionNormalize(
  D3DXQUATERNION* pOut,
  CONST D3DXQUATERNION* pQ
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pQ
[in] Pointer to the source **D3DXQUATERNION** structure.

Return Values

Pointer to a **D3DXQUATERNION** structure that is the normal of the quaternion.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionNormalize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionInverse

D3DXQuaternionRotationAxis

Rotates a quaternion about an arbitrary axis.

```
D3DXQUATERNION* D3DXQuaternionRotationAxis(
  D3DXQUATERNION* pOut,
  CONST D3DXVECTOR3* pV,
  FLOAT Angle
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pV
[in] Pointer to the **D3DXVECTOR3** structure that identifies the axis angle.

Angle
[in] Angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Return Values

Pointer to a **D3DXQUATERNION** structure rotated around the specified axis.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionRotationAxis** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionRotationMatrix, **D3DXQuaternionRotationYawPitchRoll**

D3DXQuaternionRotationMatrix

Builds a quaternion from a rotation matrix.

```
D3DXQUATERNION* D3DXQuaternionRotationMatrix(
  D3DXQUATERNION* pOut,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to the **D3DXQUATERNION** structure built from a rotation matrix.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionRotationMatrix** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionRotationAxis, **D3DXQuaternionRotationYawPitchRoll**

D3DXQuaternionRotationYawPitchRoll

Builds a quaternion with the given yaw, pitch, and roll.

```
D3DXQUATERNION* D3DXQuaternionRotationYawPitchRoll(
  D3DXQUATERNION* pOut,
  FLOAT Yaw,
  FLOAT Pitch,
  FLOAT Roll
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

Yaw
[in] Yaw around the y-axis, in radians.

Pitch
[in] Pitch around the x-axis, in radians.

Roll
[in] Roll around the z-axis, in radians.

Return Values

Pointer to a **D3DXQUATERNION** structure with the specified yaw, pitch, and roll.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionRotationYawPitchRoll** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionRotationAxis, **D3DXQuaternionRotationMatrix**

D3DXQuaternionSlerp

Interpolates between two quaternions, using spherical linear interpolation.

```
D3DXQUATERNION* D3DXQuaternionSlerp(
  D3DXQUATERNION* pOut,
  CONST D3DXQUATERNION* pQ1,
  CONST D3DXQUATERNION* pQ2,
  FLOAT t
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pQ1
[in] Pointer to a source **D3DXQUATERNION** structure.

pQ2
[in] Pointer to a source **D3DXQUATERNION** structure.

t
[in] Parameter that indicates how far to interpolate between the quaternions.

Return Values

Pointer to a **D3DXQUATERNION** structure that is the result of the interpolation.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionSlerp** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXQuaternionSquad

Interpolates between quaternions, using spherical quadrangle interpolation.

```
D3DXQUATERNION* D3DXQuaternionSquad(
  D3DXQUATERNION* pOut,
  CONST D3DXQUATERNION* pQ1,
  CONST D3DXQUATERNION* pQ2,
  CONST D3DXQUATERNION* pQ3,
  CONST D3DXQUATERNION* pQ4,
  FLOAT t
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXQUATERNION** structure that is the result of the operation.

pQ1
[in] Pointer to a source **D3DXQUATERNION** structure.

pQ2
[in] Pointer to a source **D3DXQUATERNION** structure.

pQ3
[in] Pointer to a source **D3DXQUATERNION** structure.

pQ4
[in] Pointer to a source **D3DXQUATERNION** structure.

t
[in] Parameter that indicates how far to interpolate between the quaternions.

Return Values

Pointer to a **D3DXQUATERNION** structure that is the result of the spherical quadrangle interpolation.

Remarks

This function uses the following sequence of spherical linear interpolation operations: Slerp(Slerp(Q1, Q4, t), Slerp(Q2, Q3, t), 2t(1-t))

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXQuaternionSquad** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXQuaternionExp, **D3DXQuaternionLn**

D3DXQuaternionToAxisAngle

Computes a quaternion's axis and angle of rotation.

```
void D3DXQuaternionToAxisAngle(
  CONST D3DXQUATERNION* pQ,
  D3DXVECTOR3* pAxis,
  FLOAT* pAngle
);
```

Parameters

pQ
[in, out] Pointer to the source **D3DXQUATERNION** structure.

pAxis
[in, out] When this function returns, pointer to a **D3DXVECTOR3** structure that identifies the quaternion's axis.

pAngle
[in, out] When this function returns, pointer to a **FLOAT** value that identifies the quaternion's angle of rotation in radians.

Return Values

None.

Remarks

This function expects unit quaternions.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec2Add

Adds two 2-D vectors.

```
D3DXVECTOR2* D3DXVec2Add(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR2** structure.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure.

Return Values

Pointer to a **D3DXVECTOR2** structure that is the sum of the two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2Add** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2Subtract, **D3DXVec2Scale**

D3DXVec2BaryCentric

Returns a point in Barycentric coordinates, using the specified 2-D vectors.

```
D3DXVECTOR2* D3DXVec2BaryCentric(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pV2,
  D3DXVECTOR2* pV3,
  FLOAT f,
  FLOAT g
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR2** structure.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure.

pV3
[in] Pointer to a source **D3DXVECTOR2** structure.

f
[in] Weighting factor. See Remarks.

g
[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXVECTOR2** structure in Barycentric coordinates.

Remarks

The **D3DXVec2BaryCentric** function provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function

returns the resulting point by using the following equation: V1 + f(V2-V1) + g(V3-V1).

Any point in the plane V1V2V3 can be represented by the Barycentric coordinate (*f*,*g*).The parameter *f* controls how much V2 gets weighted into the result, and the parameter *g* controls how much V3 gets weighted into the result. Lastly, 1-*f*-*g* controls how much V1 gets weighted into the result.

Note the following relations.

- If (f>=0 && g>=0 && 1-f-g>=0), the point is inside the triangle V1V2V3.
- If (f==0 && g>=0 && 1-f-g>=0), the point is on the line V1V3.
- If (f>=0 && g==0 && 1-f-g>=0), the point is on the line V1V2.
- If (f>=0 && g>=0 && 1-f-g==0), the point is on the line V2V3.

Barycentric coordinates are a form of general coordinates. In this context, using Barycentric coordinates simply represents a change in coordinate systems. What holds true for Cartesian coordinates holds true for Barycentric coordinates.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2BaryCentric** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec2CatmullRom

Performs a Catmull-Rom interpolation, using the specified 2-D vectors.

```
D3DXVECTOR2* D3DXVec2CatmullRom(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pV2,
  CONST D3DXVECTOR2* pV3,
  CONST D3DXVECTOR2* pV4,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR2** structure, a position vector.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure, a position vector.

pV3
[in] Pointer to a source **D3DXVECTOR2** structure, a position vector.

pV4
[[in] Pointer to a source **D3DXVECTOR2** structure, a position vector.

s
[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXVECTOR2** structure that is the result of the Catmull-Rom interpolation.

Remarks

The **D3DXVec2CatmullRom** function interpolates between the position V1 when *s* is equal to 0, and between the position V2 when *s* is equal to 1, using Catmull-Rom interpolation.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec2CCW

Returns the z-component by taking the cross product of two 2-D vectors.

```
FLOAT D3DXVec2CCW(
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pV2
);
```

Parameters

pV1
[in] Pointer to a source **D3DXVECTOR2** structure.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure.

Return Values

The z-component.

Remarks

This function determines the z-component by determining the cross-product based on the following formula: ((x1,y1,0) cross (x2,y2,0)). Or as shown in the following example.

```
pV1->x * pV2->y - pV1->y * pV2->x
```

If the value of the z-component is positive, the vector V2 is counter-clockwise from the vector V1. This information is useful for back-face culling.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2Dot

D3DXVec2Dot

Determines the dot-product of two 2-D vectors.

```
FLOAT D3DXVec2Dot(
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pV2
);
```

Parameters

pV1
[in] Pointer to a source **D3DXVECTOR2** structure.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure.

Return Values

The dot-product.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2CCW

D3DXVec2Hermite

Performs a Hermite spline interpolation, using the specified 2-D vectors.

```
D3DXVECTOR2* D3DXVec2Hermite(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pT1,
  CONST D3DXVECTOR2* pV2,
  CONST D3DXVECTOR2* pT2,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR2** structure, a position vector.

pT1
[in] Pointer to a source **D3DXVECTOR2** structure, a tangent vector.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure, a position vector.

pT2
[in] Pointer to a source **D3DXVECTOR2** structure, a tangent vector.

s
[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXVECTOR2** structure that is the result of the Hermite spline interpolation.

Remarks

The **D3DXVec2Hermite** function interpolates from (positionA, tangentA) to (positionB, tangentB) using Hermite spline interpolation. This function interpolates between the position V1 and the tangent T1, when *s* is equal to 0, and between the position V2 and the tangent T2, when *s* is equal to 1.

Hermite splines are useful for controlling animation because the curve runs through all the control points. Also, because the position and tangent are explicitly specified at the ends of each segment, it is easy to create a C2 continuous curve as long as you make sure that your starting position and tangent match the ending values of the last segment.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2Hermite** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec2Length

Returns the length of a 2-D vector.

```
FLOAT D3DXVec2Length(
  CONST D3DXVECTOR2* pV
);
```

Parameters

pV
[in] Pointer to the source **D3DXVECTOR2** structure.

Return Values

The vector's length.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2LengthSq

D3DXVec2LengthSq

Returns the square of the length of a 2-D vector.

```
FLOAT D3DXVec2LengthSq(
  CONST D3DXVECTOR2* pV
);
```

Parameters

pV
[in] Pointer to the source **D3DXVECTOR2** structure.

Return Values

The vector's squared length.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2Length

D3DXVec2Lerp

Performs a linear interpolation between two 2-D vectors.

```
D3DXVECTOR2* D3DXVec2Lerp(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pV2,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR2** structure.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure.

s
[in] Parameter that linearly interpolates between the vectors.

Return Values

Pointer to a **D3DXVECTOR2** structure that is the result of the linear interpolation.

Remarks

This function performs the linear interpolation based on the following formula: V1 + s(V2-V1).

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2Lerp** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec2Maximize

Returns a 2-D vector that is made up of the largest components of two 2-D vectors.

```
D3DXVECTOR2* D3DXVec2Maximize(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR2** structure.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure.

Return Values

Pointer to a **D3DXVECTOR2** structure that is made up of the largest components of the two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2Maximize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2Minimize

D3DXVec2Minimize

Returns a 2-D vector that is made up of the smallest components of two 2-D vectors.

```
D3DXVECTOR2* D3DXVec2Minimize(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR2** structure.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure.

Return Values

Pointer to a **D3DXVECTOR2** structure that is made up of the smallest components of the two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2Minimize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2Maximize

D3DXVec2Normalize

Returns the normalized version of a 2-D vector.

```
D3DXVECTOR2* D3DXVec2Normalize(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR2** structure.

Return Values

Pointer to a **D3DXVECTOR2** structure that is the normalized version of the vector.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2Normalize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec2Scale

Scales a 2-D vector.

```
D3DXVECTOR2* D3DXVec2Scale(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR2** structure.

s
[in] Scaling value.

Return Values

Pointer to a **D3DXVECTOR2** structure that is the scaled vector.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2Scale** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2Add, **D3DXVec2Subtract**

D3DXVec2Subtract

Subtracts two 2-D vectors.

```
D3DXVECTOR2* D3DXVec2Subtract(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV1,
  CONST D3DXVECTOR2* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR2** structure.

pV2
[in] Pointer to a source **D3DXVECTOR2** structure.

Return Values

Pointer to a **D3DXVECTOR2** structure that is the difference of two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2Subtract** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2Add, **D3DXVec2Scale**

D3DXVec2Transform

Transforms a 2-D vector by a given matrix.

```
D3DXVECTOR4* D3DXVec2Transform(
  D3DXVECTOR4** pOut,
  CONST D3DXVECTOR2* pV,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR2** structure.

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the transformed vector.

Remarks

This function transforms the vector, *pV* (x, y, 0, 1), by the matrix, *pM*.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2Transform** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2TransformCoord, **D3DXVec2TransformNormal**

D3DXVec2TransformCoord

Transforms a 2-D vector by a given matrix, projecting the result back into w = 1.

```
D3DXVECTOR2* D3DXVec2TransformCoord(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR2** structure.

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXVECTOR2** structure that is the transformed vector.

Remarks

This function transforms the vector, *pV* (x, y, 0, 1), by the matrix, *pM*, projecting the result back into w=1.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2TransformCoord** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2Transform, **D3DXVec2TransformNormal**

D3DXVec2TransformNormal

Transforms the 2-D vector normal by the given matrix.

```
D3DXVECTOR2* D3DXVec2TransformNormal(
  D3DXVECTOR2* pOut,
  CONST D3DXVECTOR2* pV,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR2** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR2** structure.

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXVECTOR2** structure that is the transformed vector.

Remarks

This function transforms the vector normal (x, y, 0, 0) of the vector, *pV*, by the matrix, *pM*.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec2TransformNormal** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec2Transform, **D3DXVec2TransformCoord**

D3DXVec3Add

Adds two 3-D vectors.

```
D3DXVECTOR3* D3DXVec3Add(
  D3DXVECTOR3* pOut,
```

```
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the sum of the two 3-D vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Add** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Subtract, **D3DXVec3Scale**

D3DXVec3BaryCentric

Returns a point in Barycentric coordinates, using the specified 3-D vectors.

```
D3DXVECTOR3* D3DXVec3BaryCentric(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2,
  D3DXVECTOR3* pV3,
  FLOAT f,
  FLOAT g
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure.

pV3
[in] Pointer to a source **D3DXVECTOR3** structure.

f
[in] Weighting factor. See Remarks.

g
[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXVECTOR3** structure in Barycentric coordinates.

Remarks

The **D3DXVec3BaryCentric** function provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function returns the resulting point by using the following equation: V1 + f(V2-V1) + g(V3-V1).

Any point in the plane V1V2V3 can be represented by the Barycentric coordinate (*f*,*g*).The parameter *f* controls how much V2 gets weighted into the result and the parameter *g* controls how much V3 gets weighted into the result. Lastly, 1-*f*-*g* controls how much V1 gets weighted into the result.

Note the following relations.

- If (f>=0 && g>=0 && 1-f-g>=0), the point is inside the triangle V1V2V3.
- If (f==0 && g>=0 && 1-f-g>=0), the point is on the line V1V3.
- If (f>=0 && g==0 && 1-f-g>=0), the point is on the line V1V2.
- If (f>=0 && g>=0 && 1-f-g==0), the point is on the line V2V3.

Barycentric coordinates are a form of general coordinates. In this context, using Barycentric coordinates simply represents a change in coordinate systems. What holds true for Cartesian coordinates holds true for Barycentric coordinates.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3BaryCentric** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec3CatmullRom

Performs a Catmull-Rom interpolation, using the specified 3-D vectors.

```
D3DXVECTOR3* D3DXVec3CatmullRom(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2,
  CONST D3DXVECTOR3* pV3,
  CONST D3DXVECTOR3* pV4,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure, a position vector.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure, a position vector.

pV3
[in] Pointer to a source **D3DXVECTOR3** structure, a position vector.

pV4
[[in] Pointer to a source **D3DXVECTOR3** structure, a position vector.

s
[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the result of the Catmull-Rom interpolation.

Remarks

The **D3DXVec3CatmullRom** function interpolates between the position V1 when *s* is equal to 0, and between the position V2 when *s* is equal to 1, using Catmull-Rom interpolation.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec3Cross

Determines the cross-product of two 3-D vectors.

```
D3DXVECTOR3* D3DXVec3Cross(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the cross product of two 3-D vectors.

Remarks

This function determines the cross-product with the following code.

```
D3DXVECTOR3 v;

v.x = pV1->y * pV2->z - pV1->z * pV2->y;
v.y = pV1->z * pV2->x - pV1->x * pV2->z;
v.z = pV1->x * pV2->y - pV1->y * pV2->x;

*pOut = v;
```

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Cross** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Dot

D3DXVec3Dot

Determines the dot-product of two 3-D vectors.

```
FLOAT D3DXVec3Dot(
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2
);
```

Parameters

pV1
[in] Pointer to a source **D3DXVECTOR3** structure.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure.

Return Values

The dot-product.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Cross

D3DXVec3Hermite

Performs a Hermite spline interpolation, using the specified 3-D vectors.

```
D3DXVECTOR3* D3DXVec3Hermite(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pT1,
  CONST D3DXVECTOR3* pV2,
  CONST D3DXVECTOR3* pT2,
```

```
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure, a position vector.

pT1
[in] Pointer to a source **D3DXVECTOR3** structure, a tangent vector.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure, a position vector.

pT2
[in] Pointer to a source **D3DXVECTOR3** structure, a tangent vector.

s
[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the result of the Hermite spline interpolation.

Remarks

The **D3DXVec3Hermite** function interpolates from (positionA, tangentA) to (positionB, tangentB) using Hermite spline interpolation. This function interpolates between the position V1 and the tangent T1, when *s* is equal to zero, and between the position V2 and the tangent T2, when *s* is equal to one.

Hermite splines are useful for controlling animation because the curve runs through all the control points. Also, because the position and tangent are explicitly specified at the ends of each segment, it is easy to create a C2 continuous curve as long as you make sure that your starting position and tangent match the ending values of the last segment.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Hermite** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec3Length

Returns the length of a 3-D vector.

```
FLOAT D3DXVec3Length(
  CONST D3DXVECTOR3* pV
);
```

Parameters

pV
[in] Pointer to the source **D3DXVECTOR3** structure.

Return Values

The vector's length.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3LengthSq

D3DXVec3LengthSq

Returns the square of the length of a 3-D vector.

```
FLOAT D3DXVec3LengthSq(
  CONST D3DXVECTOR3* pV
);
```

Parameters

pV
[in] Pointer to the source **D3DXVECTOR3** structure.

Return Values

The vector's squared length.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Length

D3DXVec3Lerp

Performs a linear interpolation between two 3-D vectors.

```
D3DXVECTOR3* D3DXVec3Lerp(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure.

s
[in] Parameter that linearly interpolates between the vectors.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the result of the linear interpolation.

Remarks

This function performs the linear interpolation based on the following formula: V1 + s(V2-V1).

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Lerp** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec3Maximize

Returns a 3-D vector that is made up of the largest components of two 3-D vectors.

```
D3DXVECTOR3* D3DXVec3Maximize(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure.

Return Values

Pointer to a **D3DXVECTOR3** structure that is made up of the largest components of the two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Maximize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Minimize

D3DXVec3Minimize

Returns a 3-D vector that is made up of the smallest components of two 3-D vectors.

```
D3DXVECTOR3* D3DXVec3Minimize(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure.

Return Values

Pointer to a **D3DXVECTOR3** structure that is made up of the smallest components of the two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Minimize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Maximize

D3DXVec3Normalize

Returns the normalized version of a 3-D vector.

```
D3DXVECTOR3* D3DXVec3Normalize(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR3** structure.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the normalized version of the specified vector.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Normalize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec3Project

Projects a vector from object space into screen space.

```
D3DXVECTOR3* D3DXVec3Project(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV
  CONST D3DVIEWPORT8* pViewport,
  CONST D3DXMATRIX* pProjection
  CONST D3DXMATRIX* pView,
  CONST D3DXMATRIX* pWorld
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR3** structure.

pViewport
[in] Pointer to a **D3DVIEWPORT8** structure, representing the viewport.

pProjection
[in] Pointer to a **D3DXMATRIX** structure, representing the projection matrix.

pView
[in] Pointer to a **D3DXMATRIX** structure, representing the view matrix.

pWorld
[in] Pointer to a **D3DXMATRIX** structure, representing the world matrix.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the vector projected from object space to screen space.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Project** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Unproject

D3DXVec3Scale

Scales a 3-D vector.

```
D3DXVECTOR3* D3DXVec3Scale(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR3** structure.

s
[in] Scaling value.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the scaled vector.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Scale** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Add, **D3DXVec3Subtract**

D3DXVec3Subtract

Subtracts two 3-D vectors.

```
D3DXVECTOR3* D3DXVec3Subtract(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV1,
  CONST D3DXVECTOR3* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR3** structure.

pV2
[in] Pointer to a source **D3DXVECTOR3** structure.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the difference of two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Subtract** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Add, **D3DXVec3Scale**

D3DXVec3Transform

Transforms a 3-D vector by a given matrix.

```
D3DXVECTOR4* D3DXVec3Transform(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR3* pV,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR3** structure.

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the transformed vector.

Remarks

This function transforms the vector, *pV* (x, y, z, 1), by the matrix, *pM*.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Transform** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3TransformCoord, **D3DXVec3TransformNormal**

D3DXVec3TransformCoord

Transforms a 3-D vector by a given matrix, projecting the result back into w = 1.

```
D3DXVECTOR3* D3DXVec3TransformCoord(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR3** structure.

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the transformed vector.

Remarks

This function transforms the vector, *pV* (x, y, z, 1), by the matrix, *pM*, projecting the result back into w=1.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3TransformCoord** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Transform, **D3DXVec3TransformNormal**

D3DXVec3TransformNormal

Transforms the 3-D vector normal by the given matrix.

```
D3DXVECTOR3* D3DXVec3TransformNormal(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR3** structure.

pM
[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the transformed vector.

Remarks

This function transforms the vector normal (x, y, z, 0) of the vector, *pV*, by the matrix, *pM*.

If you transform a normal by a non-affine matrix, the matrix you pass to this function should be the transpose of the inverse of the matrix you would use to transform a coordinate.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3TransformNormal** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec3Unproject

Projects a vector from screen space into object space.

```
D3DXVECTOR3* D3DXVec3Unproject(
  D3DXVECTOR3* pOut,
  CONST D3DXVECTOR3* pV
```

```
  CONST D3DVIEWPORT8* pViewport,
  CONST D3DXMATRIX* pProjection
  CONST D3DXMATRIX* pView,
  CONST D3DXMATRIX* pWorld
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR3** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR3** structure.

pViewport
[in] Pointer to a **D3DVIEWPORT8** structure, representing the viewport.

pProjection
[in] Pointer to a **D3DXMATRIX** structure, representing the projection matrix.

pView
[in] Pointer to a **D3DXMATRIX** structure, representing the view matrix.

pWorld
[in] Pointer to a **D3DXMATRIX** structure, representing the world matrix.

Return Values

Pointer to a **D3DXVECTOR3** structure that is the vector projected from screen space to screen object.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec3Unproject** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Project

D3DXVec4Add

Adds two 4-D vectors.

```
D3DXVECTOR4* D3DXVec4Add(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV1,
  CONST D3DXVECTOR4* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR4** structure.

pV2
[in] Pointer to a source **D3DXVECTOR4** structure.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the sum of the two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Add** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec4Subtract, **D3DXVec4Scale**

D3DXVec4BaryCentric

Returns a point in Barycentric coordinates, using the specified 4-D vectors.

```
D3DXVECTOR4* D3DXVec4BaryCentric(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV1,
  CONST D3DXVECTOR4* pV2,
  CONST D3DXVECTOR4* pV3,
  FLOAT f,
  FLOAT g
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR4** structure.

pV2
[in] Pointer to a source **D3DXVECTOR4** structure.

pV3
[in] Pointer to a source **D3DXVECTOR4** structure.

f
[in] Weighting factor. See Remarks.

g
[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXVECTOR4** structure in Barycentric coordinates.

Remarks

The **D3DXVec4BaryCentric** function provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function returns the resulting point by using the following equation: V1 + f(V2-V1) + g(V3-V1).

Any point in the plane V1V2V3 can be represented by the Barycentric coordinate (*f*,*g*).The parameter *f* controls how much V2 gets weighted into the result and the parameter *g* controls how much V3 gets weighted into the result. Lastly, 1-*f*-*g* controls how much V1 gets weighted into the result.

Note the following relations.

- If (f>=0 && g>=0 && 1-f-g>=0), the point is inside the triangle V1V2V3.
- If (f==0 && g>=0 && 1-f-g>=0), the point is on the line V1V3.
- If (f>=0 && g==0 && 1-f-g>=0), the point is on the line V1V2.
- If (f>=0 && g>=0 && 1-f-g==0), the point is on the line V2V3.

Barycentric coordinates are a form of general coordinates. In this context, using Barycentric coordinates simply represents a change in coordinate systems. What holds true for Cartesian coordinates holds true for Barycentric coordinates.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4BaryCentric** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec4CatmullRom

Performs a Catmull-Rom interpolation, using the specified 4-D vectors.

```
D3DXVECTOR4* D3DXVec4CatmullRom(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV1,
  CONST D3DXVECTOR4* pV2,
  CONST D3DXVECTOR4* pV3,
  CONST D3DXVECTOR4* pV4,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR4** structure, a position vector.

pV2
[in] Pointer to a source **D3DXVECTOR4** structure, a position vector.

pV3
[in] Pointer to a source **D3DXVECTOR4** structure, a position vector.

pV4
[[in] Pointer to a source **D3DXVECTOR4** structure, a position vector.

s
[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the result of the Catmull-Rom interpolation.

Remarks

The **D3DXVec4CatmullRom** function interpolates between the position V1 when *s* is equal to 0, and between the position V2 when *s* is equal to 1, using Catmull-Rom interpolation.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec4Cross

Determines the cross-product in four dimensions.

```
D3DXVECTOR4* D3DXVec4Cross(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV1,
  CONST D3DXVECTOR4* pV2,
  CONST D3DXVECTOR4* pV3
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR4** structure.

pV2
[in] Pointer to a source **D3DXVECTOR4** structure.

pV3
[in] Pointer to a source **D3DXVECTOR4** structure.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the cross product.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Cross** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec4Dot

D3DXVec4Dot

Determines the dot-product of two 4-D vectors.

```
FLOAT D3DXVec4Dot(
  CONST D3DXVECTOR4* pV1,
  CONST D3DXVECTOR4* pV2
);
```

Parameters

pV1
[in] Pointer to a source **D3DXVECTOR4** structure.

pV2
[in] Pointer to a source **D3DXVECTOR4** structure.

Return Values

The dot-product.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec4Cross

D3DXVec4Hermite

Performs a Hermite spline interpolation, using the specified 4-D vectors.

```
D3DXVECTOR4* D3DXVec4Hermite(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV1,
  CONST D3DXVECTOR4* pT1,
  CONST D3DXVECTOR4* pV2,
  CONST D3DXVECTOR4* pT2,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV1

[in] Pointer to a source **D3DXVECTOR4** structure, a position vector.

pT1

[in] Pointer to a source **D3DXVECTOR4** structure, a tangent vector.

pV2

[in] Pointer to a source **D3DXVECTOR4** structure, a position vector.

pT2

[in] Pointer to a source **D3DXVECTOR4** structure, a tangent vector.

s

[in] Weighting factor. See Remarks.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the result of the Hermite spline interpolation.

Remarks

The **D3DXVec4Hermite** function interpolates from (positionA, tangentA) to (positionB, tangentB) using Hermite spline interpolation. This function interpolates between the position V1 and the tangent T1, when *s* is equal to zero, and between the position V2 and the tangent T2, when *s* is equal to one.

Hermite splines are useful for controlling animation because the curve runs through all the control points. Also, because the position and tangent are explicitly specified at the ends of each segment, it is easy to create a C2 continuous curve as long as you make sure that your starting position and tangent match the ending values of the last segment.

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Hermite** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec4Length

Returns the length of a 4-D vector.

```
FLOAT D3DXVec4Length(
  CONST D3DXVECTOR4* pV
);
```

Parameters

pV

[in] Pointer to the source **D3DXVECTOR4** structure.

Return Values

The vector's length.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec4LengthSq

D3DXVec4LengthSq

Returns the square of the length of a 4-D vector.

```
FLOAT D3DXVec4LengthSq(
  CONST D3DXVECTOR4* pV
);
```

Parameters

pV
[in] Pointer to the source **D3DXVECTOR4** structure.

Return Values

The vector's squared length.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec4Length

D3DXVec4Lerp

Performs a linear interpolation between two 4-D vectors.

```
D3DXVECTOR4* D3DXVec4Lerp(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV1,
```

```
  CONST D3DXVECTOR4* pV2,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR4** structure.

pV2
[in] Pointer to a source **D3DXVECTOR4** structure.

s
[in] Parameter that linearly interpolates between the vectors.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the result of the linear interpolation.

Remarks

This function performs the linear interpolation based on the following formula: V1 + s(V2-V1).

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Lerp** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec4Maximize

Returns a 4-D vector that is made up of the largest components of two 4-D vectors.

```
D3DXVECTOR4* D3DXVec4Maximize(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV1,
  CONST D3DXVECTOR4* pV2
);
```

Parameters

pOut

[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV1

[in] Pointer to a source **D3DXVECTOR4** structure.

pV2

[in] Pointer to a source **D3DXVECTOR4** structure.

Return Values

Pointer to a **D3DXVECTOR4** structure that is made up of the largest components of the two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Maximize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec4Minimize

D3DXVec4Minimize

Returns a 4-D vector that is made up of the smallest components of two 4-D vectors.

```
D3DXVECTOR4* D3DXVec4Minimize(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV1,
  CONST D3DXVECTOR4* pV2
);
```

Parameters

pOut

[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV1

[in] Pointer to a source **D3DXVECTOR4** structure.

pV2

[in] Pointer to a source **D3DXVECTOR4** structure.

Return Values

Pointer to a **D3DXVECTOR4** structure that is made up of the smallest components of the two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Minimize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec4Maximize

D3DXVec4Normalize

Returns the normalized version of a 4-D vector.

```
D3DXVECTOR4* D3DXVec4Normalize(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR4** structure.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the normalized version of the vector.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Normalize** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

D3DXVec4Scale

Scales a 4-D vector.

```
D3DXVECTOR4* D3DXVec4Scale(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV,
  FLOAT s
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV
[in] Pointer to the source **D3DXVECTOR4** structure.

s
[in] Scaling value.

Return Values

Pointer to the **D3DXVECTOR4** structure that is the scaled vector.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Scale** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec4Add, **D3DXVec4Subtract**

D3DXVec4Subtract

Subtracts two 4-D vectors.

```
D3DXVECTOR4* D3DXVec4Subtract(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV1,
  CONST D3DXVECTOR4* pV2
);
```

Parameters

pOut
[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV1
[in] Pointer to a source **D3DXVECTOR4** structure.

pV2
[in] Pointer to a source **D3DXVECTOR4** structure.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the difference of two vectors.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Subtract** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

See Also

D3DXVec3Add, **D3DXVec3Scale**

D3DXVec4Transform

Transforms a 4-D vector by a given matrix.

```
D3DXVECTOR4* D3DXVec4Transform(
  D3DXVECTOR4* pOut,
  CONST D3DXVECTOR4* pV,
  CONST D3DXMATRIX* pM
);
```

Parameters

pOut

[in, out] Pointer to the **D3DXVECTOR4** structure that is the result of the operation.

pV

[in] Pointer to the source **D3DXVECTOR4** structure.

pM

[in] Pointer to the source **D3DXMATRIX** structure.

Return Values

Pointer to a **D3DXVECTOR4** structure that is the transformed 4-D vector.

Remarks

The return value for this function is the same value returned in the *pOut* parameter. In this way, the **D3DXVec4Transform** function can be used as a parameter for another function.

Requirements

Header: Declared in D3dx8math.h.
Import Library: Use D3dx8.lib.

Mesh Functions

Direct3DX supplies the following mesh functions.

- **D3DXBoxBoundProbe**
- **D3DXCleanMesh**
- **D3DXComputeBoundingBox**
- **D3DXComputeBoundingSphere**
- **D3DXComputeNormals**
- **D3DXCreateBuffer**
- **D3DXCreateMesh**
- **D3DXCreateMeshFVF**
- **D3DXCreatePMeshFromStream**
- **D3DXCreateSkinMesh**
- **D3DXCreateSkinMeshFromMesh**
- **D3DXCreateSkinMeshFVF**
- **D3DXCreateSPMesh**
- **D3DXDeclaratorFromFVF**
- **D3DXFVFFromDeclarator**
- **D3DXGeneratePMesh**

- **D3DXIntersect**
- **D3DXLoadMeshFromX**
- **D3DXLoadMeshFromXof**
- **D3DXLoadSkinMeshFromXof**
- **D3DXSaveMeshToX**
- **D3DXSimplifyMesh**
- **D3DXSphereBoundProbe**
- **D3DXTesselateMesh**
- **D3DXValidMesh**
- **D3DXWeldVertices**

D3DXBoxBoundProbe

Determines if a ray intersects the volume of a box's bounding box.

```
BOOL D3DXBoxBoundProbe(
  CONST D3DXVECTOR3* pMin,
  CONST D3DXVECTOR3* pMax,
  D3DXVECTOR3* pRayPosition,
  D3DXVECTOR3* pRayDirection
);
```

Parameters

pMin
[out] Pointer to a **D3DXVECTOR3** structure, describing the lower-left corner of the bounding box. See Remarks.

pMax
[out] Pointer to a **D3DXVECTOR3** structure, describing the upper-right corner of the bounding box. See Remarks.

pRayPosition
[in] Pointer to a **D3DXVECTOR3** structure, specifying the origin coordinate of the ray.

pRayDirection
[in] Pointer to a **D3DXVECTOR3** structure, specifying the direction of the ray.

Return Values

Returns TRUE if the ray intersects the volume of the box's bounding box. Otherwise, returns FALSE.

Remarks

D3DXBoxBoundProbe determines if the ray intersects the volume of the box's bounding box, not just the surface of the box.

The **D3DXVECTOR3** values passed to **D3DXBoxBoundProbe** are xmin, xmax, ymin, ymax, zmin, and zmax. Thus, the following defines the corners of the bounding box.

```
xmax, ymax, zmax
xmax, ymax, zmin
xmax, ymin, zmax
xmax, ymin, zmin
xmin, ymax, zmax
xmin, ymax, zmin
xmin, ymin, zmax
xmin, ymin, zmin
```

The depth of the bounding box in the z direction is zmax - zmin, in the y direction is ymax - ymin, and in the x direction is xmax - xmin. For example, with the following minimum and maximum vectors, min (-1, -1, -1) and max (1, 1, 1), the bounding box is defined in the following manner.

```
 1,  1,  1
 1,  1, -1
 1, -1,  1
 1, -1, -1
-1,  1,  1
-1,  1, -1
-1, -1,  1
-1, -1, -1
```

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

D3DXComputeBoundingBox

D3DXCleanMesh

Cleans a mesh, preparing it for simplification.

```
HRESULT D3DXCleanMesh(
  LPD3DXMESH pMeshIn,
  CONST DWORD* pAdjacency,
```

LPD3DXMESH* *ppMeshOut*
);

Parameters

pMeshIn
[in] Pointer to an **ID3DXMesh** interface, representing the mesh to be cleaned.

pAdjacency
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh to be cleaned.

ppMeshOut
[out] An address of a pointer to an **ID3DXMesh** interface, representing the returned cleaned mesh. The same mesh is returned that was passed in if no cleaning was necessary.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

This method cleans the mesh by adding another vertex where two fans of triangles share the same vertex.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXComputeBoundingBox

Computes a bounding box.

HRESULT D3DXComputeBoundingBox(
PVOID *pPointsFVF*,
DWORD *NumVertices*,
DWORD *FVF*,
D3DXVECTOR3* *pMin*,
D3DXVECTOR3* *pMax*
);

Parameters

pPointsFVF
[in] Pointer to a buffer containing the vertex data around which to calculate the bounding box.

NumVertices
[in] Number of vertices.

FVF
[in] Combination of flexible vertex format flags that describes the vertex format.

pMin
[out] Pointer to a **D3DXVECTOR3** structure, describing the returned lower-left corner of the bounding box. See remarks.

pMax
[out] Pointer to a **D3DXVECTOR3** structure, describing the returned upper-right corner of the bounding box. See remarks.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one D3DERR_INVALIDCALL.

Remarks

The **D3DXVECTOR3** values returned by **D3DXComputeBoundingBox** are xmin, xmax, ymin, ymax, zmin, and zmax. Thus, the following defines the corners of the bounding box.

```
xmax, ymax, zmax
xmax, ymax, zmin
xmax, ymin, zmax
xmax, ymin, zmin
xmin, ymax, zmax
xmin, ymax, zmin
xmin, ymin, zmax
xmin, ymin, zmin
```

The depth of the bounding box in the z direction is zmax - zmin, in the y direction is ymax - ymin, and in the x direction is xmax - xmin. For example, with the following minimum and maximum vectors, min (-1, -1, -1) and max (1, 1, 1), the bounding box is defined in the following manner.

```
 1,  1,  1
 1,  1, -1
 1, -1,  1
 1, -1, -1
-1,  1,  1
```

-1, 1, -1
-1, -1, 1
-1, -1, -1

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXComputeBoundingSphere

Computes a bounding sphere.

```
HRESULT D3DXComputeBoundingSphere(
  PVOID pvPointsFVF,
  DWORD NumVertices,
  DWORD FVF,
  D3DXVECTOR3* pCenter,
  FLOAT* pRadius
);
```

Parameters

pvPointsFVF
[in] Pointer to a buffer containing the vertex data around which to calculate the bounding sphere.

NumVertices
[in] Number of vertices.

FVF
[in] Combination of flexible vertex format flags that describes the vertex format.

pCenter
[out] **D3DXVECTOR3** structure, defining the coordinate center of the returned bounding sphere.

pRadius
[out] Radius of the returned bounding sphere.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one D3DERR_INVALIDCALL.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXComputeNormals

Computes normals for each vertex in a mesh.

```
HRESULT D3DXComputeNormals(
  LPD3DXBASEMESH pMesh
);
```

Parameters

pMesh
[in, out] Pointer to an **ID3DXBaseMesh** interface, representing the normalized mesh object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one D3DERR_INVALIDCALL.

Note

The input mesh must have the D3DFVF_NORMAL flag specified in its FVF. A normal for a vertex is generated by averaging the normals of all faces that share that vertex.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXCreateBuffer

Creates a buffer object.

```
HRESULT D3DXCreateBuffer(
  DWORD NumBytes,
  LPD3DXBUFFER* ppBuffer
);
```

Parameters

NumBytes
[in] Size of the buffer to create, in bytes.

ppBuffer
[out] Address of a pointer to an **ID3DXBuffer** interface, representing the created buffer object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one E_OUTOFMEMORY.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXCreateMesh

Creates a mesh object using a declarator.

```
HRESULT D3DXCreateMesh(
  DWORD NumFaces,
  DWORD NumVertices,
  DWORD Options,
  CONST DWORD* pDeclaration,
  LPDIRECT3DDEVICE8 pDevice,
  LPD3DXMESH* ppMesh
);
```

Parameters

NumFaces
[in] Number of faces for the mesh. This parameter must be greater than 0.

NumVertices
[in] Number of vertices for the mesh. This parameter must be greater than 0.

Options
[in] A combination of one or more flags, specifying options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES

Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pDeclaration
[in] Pointer to a **DWORD** value, representing the declarator that describes the vertex format for the returned mesh. This parameter must map directly to an FVF.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device object to be associated with the mesh.

ppMesh
[out] Address of a pointer to an **ID3DXMesh** interface, representing the created mesh object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

D3DXDeclaratorFromFVF

D3DXCreateMeshFVF

Creates a mesh object using a flexible vertex format (FVF) code.

```
HRESULT D3DXCreateMeshFVF(
  DWORD NumFaces,
  DWORD NumVertices,
  DWORD Options,
  DWORD FVF,
  LPDIRECT3DDEVICE8 pDevice,
  LPD3DXMESH* ppMesh
);
```

Parameters

NumFaces
[in] Number of faces for the mesh. This parameter must be greater than 0.

NumVertices
[in] Number of vertices for the mesh. This parameter must be greater than 0.

Options
[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

FVF
[in] Combination of flexible vertex format flags that describes the vertex format for the returned mesh.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device object to be associated with the mesh.

ppMesh
[out] Address of a pointer to an **ID3DXMesh** interface, representing the created mesh object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

D3DXFVFFromDeclarator

D3DXCreatePMeshFromStream

Creates a progressive mesh from a stream object.

```
HRESULT D3DXCreatePMeshFromStream(
  IStream* pStream,
  DWORD Options,
  LPDIRECT3DDEVICE8 pD3DDevice,
  LPD3DXBUFFER* ppMaterials,
  DWORD* pNumMaterials,
  LPD3DXPMESH* ppPMesh
);
```

Parameters

pStream
[in] Pointer to an **IStream** interface, representing the stream object from which to create the progressive mesh. The data in this stream was generated with the **ID3DXPMesh::Save** method.

Options
[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT

The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pD3DDevice

[in] Pointer to an **IDirect3DDevice8** interface, the device object to be associated with the progressive mesh.

ppMaterials
[out] Address of a pointer to an **ID3DXBuffer** interface. When this method returns, this parameter is filled with an array of **D3DXMATERIAL** structures, containing information saved in the stream object.

pNumMaterials
[out] Pointer to the number of **D3DXMATERIAL** structures in the *ppMaterials* array, when the method returns.

ppPMesh
[out] Address of a pointer to an **ID3DXPMesh** interface, representing the created progressive mesh.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

For more information on the **IStream** interface, see the Microsoft Platform Software Development Kit (SDK).

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXPMesh::Save

D3DXCreateSkinMesh

Creates an empty skin mesh object using a declarator.

HRESULT D3DXCreateSkinMesh(
DWORD *numFaces*,
DWORD *numVertices*,
DWORD *numBones*,
DWORD *Options*,

CONST DWORD* *pDeclaration*,
LPDIRECT3DDEVICE8 *pDevice*,
LPD3DXSKINMESH* *ppSkinMesh*
);

Parameters

numFaces
[in] Number of faces for the skin mesh.

numVertices
[in] Number of vertices for the skin mesh.

numBones
[in] Number of bones for the skin mesh.

Options
[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pDeclaration
[in] Pointer to a **DWORD** value, representing the declarator that describes the vertex format for the returned skin mesh.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device object to be associated with the skin mesh.

ppSkinMesh
[out, retval] Address of a pointer to an **ID3DXSkinMesh** interface, representing the created skin mesh object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be E_OUTOFMEMORY.

Remarks

Use **ID3DXSkinMesh::SetBoneInfluence** to populate the empty skin mesh object returned by this method.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::SetBoneInfluence

D3DXCreateSkinMeshFromMesh

Creates a skin mesh from another mesh.

```
HRESULT D3DXCreateSkinMeshFromMesh(
  LPD3DXMESH pMesh,
  DWORD numBones,
  LPD3DXSKINMESH* ppSkinMesh
);
```

Parameters

pMesh
[in] Pointer to an **ID3DXMesh** object, the mesh from which to create the skin mesh.

numBones
[in] Number of bones for the skin mesh.

ppSkinMesh
[out, retval] Address of a pointer to an **ID3DXSkinMesh** interface, representing the created skin mesh object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be E_OUTOFMEMORY.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXCreateSkinMeshFVF

Creates an empty skin mesh object using a flexible vertex format (FVF) code.

```
HRESULT D3DXCreateSkinMeshFVF(
  DWORD numFaces,
  DWORD numVertices,
  DWORD numBones,
  DWORD Options,
  DWORD FVF,
  LPDIRECT3DDEVICE8 pDevice,
  LPD3DXSKINMESH* ppSkinMesh
);
```

Parameters

numFaces
[in] Number of faces for the skin mesh.

numVertices
[in] Number of vertices for the skin mesh.

numBones
[in] Number of bones for the skin mesh.

Options
[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

FVF
[in] Combination of flexible vertex format flags that describes the vertex format for the returned skin mesh.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device object to be associated with the skin mesh.

ppSkinMesh
[out, retval] Address of a pointer to an **ID3DXSkinMesh** interface, representing the created skin mesh object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

Use **ID3DXSkinMesh::SetBoneInfluence** to populate the empty skin mesh object returned by this method.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

ID3DXSkinMesh::SetBoneInfluence

D3DXCreateSPMesh

Creates a simplification mesh.

```
HRESULT D3DXCreateSPMesh(
  LPD3DXMESH pMesh,
  CONST DWORD* pAdjacency,
  CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
  CONST FLOAT* pVertexWeights,
  LPD3DXSPMESH* ppSMesh
);
```

Parameters

pMesh
[in] Pointer to an **ID3DXMesh** interface, representing the mesh to simplify.

pAdjacency
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the created simplification mesh.

pVertexAttributeWeights
[in] Pointer to a **D3DXATTRIBUTEWEIGHTS** structure, containing the weight for each vertex component. If this parameter is set to NULL, a default structure is used. See Remarks.

pVertexWeights
[in] Pointer to an array of vertex weights. If this parameter is set to NULL, all vertex weights are set to 1.0. Note that the higher the vertex weight for a given vertex, the less likely it is to be simplified away.

ppSMesh
[out] Address of a pointer to an **ID3DXSPMesh** interface, representing the created simplification mesh.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DXERR_CANNOTATTRSORT

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

A simplification mesh is used to simplify a mesh to a lower number of triangles and faces.

If *pVertexAttributeWeights* is set to NULL, the following values are assigned to the default **D3DXATTRIBUTEWEIGHTS** structure.

```
D3DXATTRIBUTEWEIGHTS AttributeWeights;

AttributeWeights.Position  = 1.0;
AttributeWeights.Boundary =  1.0;
AttributeWeights.Normal   =  1.0;
AttributeWeights.Diffuse  =  0.0;
AttributeWeights.Specular =  0.0;
AttributeWeights.Tex[8]   =  {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
```

This default structure is what most applications should use because it considers only geometric and normal adjustment. Only in special cases will the other member fields need to be modified.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXDeclaratorFromFVF

Returns a declarator from a flexible vertex format (FVF) code.

```
HRESULT D3DXDeclaratorFromFVF(
  DWORD FVF,
  DWORD Declaration[MAX_FVF_DECL_SIZE]
);
```

Parameters

FVF
[in] Combination of flexible vertex format flags that describes the FVF from which to generate the returned declarator array.

Declaration
[out] A returned array describing the vertex format of the vertices in the queried mesh. The upper limit of this declarator array is MAX_FVF_DECL_SIZE, limiting the declarator to a maximum of 15 **DWORD**s.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one D3DXERR_INVALIDMESH.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also'

D3DXFVFFromDeclarator

D3DXFVFFromDeclarator

Returns a flexible vertex format (FVF) code from a declarator.

```
HRESULT D3DXFVFFromDeclarator(
  CONST DWORD* pDeclarator,
  DWORD* pFVF
);
```

Parameters

pDeclarator
[in] Pointer to a **DWORD** value, representing the declarator from which to generate the FVF code.

pFVF
[out] Pointer to a **DWORD** value, representing the returned combination of flexible vertex format flags that describes the vertex format returned from the declarator.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be D3DERR_INVALIDCALL.

Note

This function will fail for any declarator that doesn't map directly to an FVF.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

See Also

D3DXDeclaratorFromFVF

D3DXGeneratePMesh

Generates a progressive mesh.

```
HRESULT D3DXGeneratePMesh(
  LPD3DXMESH pMesh,
  CONST DWORD* pAdjacency,
  CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
  CONST FLOAT* pVertexWeights,
  DWORD MinValue,
  DWORD Options,
  LPD3DXPMESH* ppPMesh
);
```

Parameters

pMesh
[in] Pointer to an **ID3DXMesh** interface, representing the source mesh.

pAdjacency
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the created progressive mesh.

pVertexAttributeWeights
[in] Pointer to a **D3DXATTRIBUTEWEIGHTS** structure, containing the weight for each vertex component. If this parameter is set to NULL, a default structure is used. See Remarks.

pVertexWeights
[in] Pointer to an array of vertex weights. If this parameter is set to NULL, all vertex weights are set to 1.0. Note that the higher the vertex weight for a given vertex, the less likely it is to be simplified away.

MinValue
[in] Number of vertices or faces, depending on the flag set in the *Options* parameter, by which to simplify the source mesh.

Options
[in] Specifies simplification options for the mesh. One of the following flags can be set.

D3DXMESHSIMP_FACE
The mesh will be simplified by the number of faces specified in the *MinValue* parameter.

D3DXMESHSIMP_VERTEX
The mesh will be simplified by the number of vertices specified in the *MinValue* parameter.

ppPMesh
[out] Address of a pointer to an **ID3DXPMesh** interface, representing the created progressive mesh.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DXERR_CANNOTATTRSORT

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

This function generates a mesh where the level of detail (LOD) can be adjusted from the current value to the *MinValue*.

If the simplification process cannot reduce the mesh to *MinValue*, the call still succeeds because *MinValue*is a desired minimum, not an absolute minimum.

If *pVertexAttributeWeights* is set to NULL, the following values are assigned to the default **D3DXATTRIBUTEWEIGHTS** structure.

```
D3DXATTRIBUTEWEIGHTS AttributeWeights;

AttributeWeights.Position  = 1.0;
AttributeWeights.Boundary =  1.0;
AttributeWeights.Normal   =  1.0;
AttributeWeights.Diffuse  =  0.0;
AttributeWeights.Specular =  0.0;
AttributeWeights.Tex[8]   =  {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
```

This default structure is what most applications should use because it considers only geometric and normal adjustment. Only in special cases will the other member fields need to be modified.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXIntersect

Determines if a ray intersects with a mesh.

HRESULT D3DXIntersect(
LPD3DXBASEMESH *pMesh*,
CONST D3DXVECTOR3* *pRayPos*,
CONST D3DXVECTOR3* *pRayDir*,
BOOL* *pHit*,

DWORD* *pFaceIndex*,
FLOAT* *pU*,
FLOAT* *pV*,
FLOAT* *pDist*
);

Parameters

pMesh
[in] Pointer to an **ID3DXBaseMesh** interface, representing the mesh to be tested.

pRayPos
[in] Pointer to a **D3DXVECTOR3** structure, specifying the origin coordinate of the ray.

pRayDir
[in] Pointer to a **D3DXVECTOR3** structure, specifying the direction of the ray.

pHit
[out] Pointer to a **BOOL**. If the ray intersects a triangular face on the mesh, this value will be set to TRUE. Otherwise, this value is set to FALSE.

pFaceIndex
[out] Pointer to an index value of the face closest to the ray origin, if *pHit* is TRUE.

pU
[out] Pointer to a Barycentric hit coordinate.

pV
[out] Pointer to a Barycentric hit coordinate.

pDist
[out] Pointer to a ray intersection parameter distance.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be E_OUTOFMEMORY.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXLoadMeshFromX

Loads a mesh from a Microsoft® DirectX® (.x) file.

HRESULT D3DXLoadMeshFromX(
LPSTR *pFilename*,

DWORD *Options*,
LPDIRECT3DDEVICE8 *pDevice*,
LPD3DXBUFFER* *ppAdjacency*,
LPD3DXBUFFER* *ppMaterials*,
PDWORD *pNumMaterials*,
LPD3DXMESH* *ppMesh*
);

Parameters

pFilename
[in] Pointer to a string that specifies the name of the DirectX file to load.

Options
[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device object associated with the mesh.

ppAdjacency
[out] Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh.

ppMaterials
[out] Address of a pointer to an **ID3DXBuffer** interface. When this method returns, this parameter is filled with an array of **D3DXMATERIAL** structures, containing information saved in the Microsoft® DirectX® file.

pNumMaterials
[out] Pointer to the number of **D3DXMATERIAL** structures in the *ppMaterials* array, when the method returns.

ppMesh
[out] Address of a pointer to an **ID3DXMesh** interface, representing the loaded mesh.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Note

All the meshes in the file will be collapsed into one output mesh. If the file contains a frame hierarchy, all the transformations will be applied to the mesh.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXLoadMeshFromXof

Loads a mesh from a Microsoft® DirectX® (.x) file data object.

```
HRESULT D3DXLoadMeshFromXof(
  LPDIRECTXFILEDATA pXofObjMesh,
  DWORD Options,
  LPDIRECT3DDEVICE8 pD3DDevice,
  LPD3DXBUFFER* ppAdjacency,
  LPD3DXBUFFER* ppMaterials,
  PDWORD pNumMaterials,
  LPD3DXMESH* ppMesh
);
```

Parameters

pXofObjMesh
[in] Pointer to an **IDirectXFileData** interface, representing the Microsoft® DirectX® file data object to load.

Options
[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

D3DXMESH_MANAGED

Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pD3DDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device object associated with the mesh.

ppAdjacency
[in, out] Address of a pointer to an **ID3DXBuffer** interface. When this method returns, this parameter is filled with an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh.

ppMaterials
[in, out] Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of **D3DXMATERIAL** structures.

pNumMaterials
[in, out] Pointer to the number of **D3DXMATERIAL** structures in the *ppMaterials* array, when the method returns.

ppMesh
[out, retval] Address of a pointer to an **ID3DXMesh** interface, representing the loaded mesh.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXLoadSkinMeshFromXof

Loads a skin mesh from a Microsoft® DirectX® (.x) file data object.

```
HRESULT D3DXLoadSkinMeshFromXof(
  LPDIRECTXFILEDATA pXofObjMesh,
  DWORD Options,
  LPDIRECT3DDEVICE8 pD3DDevice,
  LPD3DXBUFFER* ppAdjacency,
  LPD3DXBUFFER* ppMaterials,
  PDWORD pMatOut,
  LPD3DXBUFFER* ppBoneNames,
  LPD3DXBUFFER* ppBoneTransforms,
  LPD3DXSKINMESH* ppMesh
);
```

Parameters

pXofObjMesh
[in] Pointer to an **IDirectXFileData** interface, representing the DirectX file data object to load.

Options
[in] A combination of one or more flags, specifying creation options for the mesh. The following flags are defined.

D3DXMESH_32BIT
The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This flag is not supported and should not be used.

D3DXMESH_DONOTCLIP
Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_DYNAMIC
Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_RTPATCHES
Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D.

D3DXMESH_MANAGED
Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_POINTS
Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_IB_MANAGED
Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_IB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_SYSTEMMEM
Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_VB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_VB_MANAGED
Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_SYSTEMMEM
Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_WRITEONLY
Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

pD3DDevice
[in] Pointer to an **IDirect3DDevice8** interface, the device object associated with the mesh.

ppAdjacency

[in, out] Address of a pointer to an **ID3DXBuffer** interface. When this method returns, this parameter is filled with an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh.

ppMaterials
[in, out] Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of **D3DXMATERIAL** structures.

pMatOut
[in, out] Pointer to the number of **D3DXMATERIAL** structures in the *ppMaterials* array, when the method returns.

ppBoneNames
[in, out] Address of a pointer to an **ID3DXBuffer** interface, containing the bone names.

ppBoneTransforms
[in, out] Address of a pointer to an **ID3DXBuffer** interface, containing the bone transforms.

ppMesh
[out, retval] Address of a pointer to an **ID3DXSkinMesh** interface, representing the loaded mesh.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

This method takes a pointer to an internal object in the .x file, enabling you to load the frame hierarchy.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXSaveMeshToX

Saves a mesh to a Microsoft® DirectX® (.x) file.

HRESULT D3DXSaveMeshToX(
LPSTR *pFilename*,

```
  LPD3DXMESH pMesh,
  CONST DWORD* pAdjacency,
  CONST LPD3DXMATERIAL pMaterials,
  DWORD NumMaterials,
  DWORD Format
);
```

Parameters

pFilename
[in] Pointer to a string that specifies the name of the DirectX file identifying the saved mesh.

pMesh
[in] Pointer to an **ID3DXMesh** interface, representing the mesh to save to a DirectX file.

pAdjacency
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh.

pMaterials
[in] Pointer to an array of **D3DXMATERIAL** structures, containing material information to be saved in the DirectX file.

NumMaterials
[in] Number of **D3DXMATERIAL** structures in the *rgMaterials array*

Format
[in] Indicates the format to use when saving the DirectX file. For more information, see Remarks.

DXFILEFORMAT_BINARY
Indicates a binary file.

DXFILEFORMAT_COMPRESSED
Indicates a compressed file.

DXFILEFORMAT_TEXT
Indicates a text file.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be D3DERR_INVALIDCALL.

Remarks

The default value for the file format is DXFILEFORMAT_BINARY. The file format values can be combined together in a logical **OR** to create compressed text or compressed binary files. If a file is specified as both binary (0) and text (1), it will be saved as a text file because the value will be indistinguishable from the text file

format value (0 + 1 = 1). If you indicate that the file format should be text and compressed, the file will first be written out as text and then compressed. However, compressed text files are not as efficient as binary text files, so in most cases you will want to indicate binary and compressed. Setting a file to be compressed without specifying a format will result in a binary, compressed file.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXSimplifyMesh

Simplifies a mesh.

```
HRESULT D3DXSimplifyMesh(
  LPD3DXMESH pMesh,
  CONST DWORD* pAdjacency,
  CONST LPD3DXATTRIBUTEWEIGHTS pVertexAttributeWeights,
  CONST FLOAT* pVertexWeights,
  DWORD MinValue,
  DWORD* Options,
  LPD3DXMESH* ppMesh
);
```

Parameters

pMesh
[in] Pointer to an **ID3DXMesh** interface, representing the source mesh.

pAdjacency
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh to be simplified.

pVertexAttributeWeights
[in] Pointer to a **D3DXATTRIBUTEWEIGHTS** structure, containing the weight for each vertex component. If this parameter is set to NULL, a default structure is used. See Remarks.

pVertexWeights
[in] Pointer to an array of vertex weights. If this parameter is set to NULL, all vertex weights are set to 1.0.

MinValue
[in] Number of vertices or faces, depending on the flag set in the *Options* parameter, by which to simplify the source mesh.

Options
[in] Specifies simplification options for the mesh. One of the following flags can be set.

D3DXMESHSIMP_FACE
The mesh will be simplified by the number of faces specified in the *MinValue* parameter.

D3DXMESHSIMP_VERTEX
The mesh will be simplified by the number of vertices specified in the *MinValue* parameter.

ppMesh
[out] Address of a pointer to an **ID3DXMesh** interface, representing the returned simplification mesh.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

This function generates a mesh that has *minValue* vertices or faces.

If the simplification process cannot reduce the mesh to *minValue*, the call still succeeds because *minValue* is a desired minimum, not an absolute minimum.

If *pVertexAttributeWeights* is set to NULL, the following values are assigned to the default **D3DXATTRIBUTEWEIGHTS** structure.

```
D3DXATTRIBUTEWEIGHTS AttributeWeights;

AttributeWeights.Position  = 1.0;
AttributeWeights.Boundary =  1.0;
AttributeWeights.Normal   =  1.0;
AttributeWeights.Diffuse  =  0.0;
AttributeWeights.Specular =  0.0;
AttributeWeights.Tex[8]   =  {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
```

This default structure is what most applications should use because it considers only geometric and normal adjustment. Only in special cases will the other member fields need to be modified.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXSphereBoundProbe

Determines if a ray intersects the volume of a sphere's bounding box.

```
BOOL D3DXSphereBoundProbe(
  CONST D3DXVECTOR3* pCenter,
  FLOAT Radius,
  D3DXVECTOR3* pRayPosition,
  D3DXVECTOR3* pRayDirection
);
```

Parameters

pCenter
[in] Pointer to a **D3DXVECTOR3** structure, specifying the center coordinate of the sphere.

Radius
[in] Radius of the sphere.

pRayPosition
[in] Pointer to a **D3DXVECTOR3** structure, specifying the origin coordinate of the ray.

pRayDirection
[in] Pointer to a **D3DXVECTOR3** structure, specifying the direction of the ray.

Return Values

Returns TRUE if the ray intersects the volume of the sphere's bounding box. Otherwise, returns FALSE.

Remarks

D3DXSphereBoundProbe determines if the ray intersects the volume of the sphere's bounding box, not just the surface of the sphere.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXTesselateMesh

Tessellates a mesh.

```
HRESULT D3DXTesselateMesh(
  LPD3DXMESH pMeshIn,
  CONST DWORD* pAdjacency,
```

```
  FLOAT NumSegs,
  BOOL bQuadraticNormals
  LPD3DXMESH* ppMeshOut
);
```

Parameters

pMeshIn
[in] Pointer to an **ID3DXMesh** interface, representing the mesh to tessellate.

pAdjacency
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the source mesh.

NumSegs
[in] Number of segments per edge to tessellate.

bQuadraticNormals
[in] If set to TRUE, use quadratic interpolation for normals. If set to FALSE, use linear interpolation.

ppMeshOut
[out] Address of a pointer to an **ID3DXMesh** interface, representing the returned tessellated mesh.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Note

This function tessellates by using the n-Patch algorithm.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXValidMesh

Validates a mesh.

```
HRESULT D3DXValidMesh(
  LPD3DXMESH pMeshIn,
```

CONST DWORD* *pAdjacency*
);

Parameters

pMeshIn
[in] Pointer to an **ID3DXMesh** interface, representing the mesh to be tested.

pAdjacency
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh to be tested.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DXERR_INVALIDMESH

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

This method validates the mesh by checking for invalid indices and adding another vertex where two fans of triangles share the same vertex.

Error information is available from the debug spew.

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

D3DXWeldVertices

Welds replicated vertices together that have attributes that are equal.

HRESULT D3DXValidMesh(
CONST LPD3DXMESH *pMesh*,
float *fEpsilon*,
CONST DWORD* *pAdjacencyIn*,
DWORD* *pAdjacencyOut*,
DWORD* *pFaceRemap*,
LPD3DXBUFFER* *ppVertexRemap*
);

Parameters

pMesh
[in] Pointer to an **ID3DXMesh** object, the mesh from which to weld vertices.

fEpsilon
[in] Difference in vertex attributes, under which vertices are welded.

pAdjacencyIn
[in] Pointer to an array of three **DWORD**s per face that specify the three neighbors for each face in the source mesh. If this parameter is set to NULL, then **ID3DXMesh::GenerateAdjacency** will be called to create logical adjacency information.

pAdjacencyOut
[in, out] Pointer to a destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

pFaceRemap
[out] Pointer to a destination buffer containing the new index for each face.

ppVertexRemap
[out] Address of a pointer to an **ID3DXBuffer** interface, containing the new index for each vertex.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

This method uses the provided adjacency information to determine the points that are replicated. This method uses an epsilon compare to merge vertices and requires vertices with equal position to already have been calculated and represented by point representative data.

This method takes logically welded vertices and combines the ones that have similar components. Such as normals within *fEpsilon* or texture coordinates within *fEpsilon.*

Requirements

Header: Declared in D3dx8mesh.h.
Import Library: Use D3dx8.lib.

Shader Functions

Direct3DX supplies the following shader functions.

- **D3DXAssembleShader**
- **D3DXAssembleShaderFromFileA**
- **D3DXAssembleShaderFromFileW**

The assembler assumes that is assembling a vertex shader (version 1.0) unless a version token is encountered at the start of the assembly. For information on the Direct3DX vertex and pixel shader assemblers, see Direct3DX Shader Assemblers Reference.

D3DXAssembleShader

Assembles an ASCII description of a shader into binary form, where the shader source is in memory.

```
HRESULT D3DXAssembleShader(
  LPCVOID pSrcData,
  UINT SrcDataLen,
  DWORD Flags,
  LPD3DXBUFFER* ppConstants,
  LPD3DXBUFFER* ppCompiledShader,
  LPD3DXBUFFER* ppCompilationErrors
);
```

Parameters

pSrcData
[in] Pointer to the source code.

SrcDataLen
[in] Size of the source code, in bytes.

Flags
[in] A combination of the following flags, specifying assembly options.

D3DXASM_DEBUG
Inserts debugging information as comments in the assembled shader.

D3DXASM_SKIPVALIDATION
Do not validate the generated code against known capabilities and constraints. This option is recommended only when assembling a shader you know will function (that is, the shader has been assembled before without this option.)

ppConstants
[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned constant declarations. These constants are returned as a vertex shader declaration fragment. It is up to the application to insert the contents of this buffer into their

declaration. For pixel shaders this parameter is meaningless because constant declarations are included in the assembled shader. This parameter is ignored if it is NULL.

ppCompiledShader
[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned compiled object code. This parameter is ignored if it is NULL.

ppCompilationErrors
[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned ASCII error messages. This parameter is ignored if it is NULL.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

D3DXAssembleShaderFromFileA

Assembles an ASCII description of a shader into binary form, where the source file name is an ANSI string.

```
HRESULT D3DXAssembleShaderFromFileA(
  LPCSTR pSrcFile,
  DWORD Flags,
  LPD3DXBUFFER* ppConstants,
  LPD3DXBUFFER* ppCompiledShader,
  LPD3DXBUFFER* ppCompilationErrors
);
```

Parameters

pSrcFile
[in] Pointer to the source file name, in ANSI format.

Flags
[in] A combination of the following flags, specifying assembly options.

D3DXASM_DEBUG

Inserts debugging information as comments in the assembled shader.

D3DXASM_SKIPVALIDATION
Do not validate the generated code against known capabilities and constraints. This option is recommended only when assembling a shader you know will function (that is, the shader has been assembled before without this option.)

ppConstants
[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned constant declarations. These constants are returned as a vertex shader declaration fragment. It is up to the application to insert the contents of this buffer into their declaration. For pixel shaders this parameter is meaningless because constant declarations are included in the assembled shader. This parameter is ignored if it is NULL.

ppCompiledShader
[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned compiled object code. This parameter is ignored if it is NULL.

ppCompilationErrors
[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned ASCII error messages. This parameter is ignored if it is NULL.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

D3DXAssembleShaderFromFile maps to either **D3DXAssembleShaderFromFileA** or **D3DXAssembleShaderFromFileW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings.

The following code fragment shows how **D3DXAssembleShaderFromFile** is defined.

```
#ifdef UNICODE
#define D3DXAssembleShaderFromFile D3DXAssembleShaderFromFileW
#else
#define D3DXAssembleShaderFromFile D3DXAssembleShaderFromFileA
#endif
```

The following example wrapper function, **PreprocessAndAssembleShaderFromFile**, shows how you can invoke the C preprocessor on your vertex shader code before invoking the assembler with **D3DXAssembleShaderFromFile**.

```
HRESULT PreprocessAndAssembleShaderFromFile(
      LPCSTR          szFile,
      DWORD           Flags,
      LPD3DXBUFFER*   ppConstants,
      LPD3DXBUFFER*   ppCode,
      LPD3DXBUFFER*   ppErrors)
{
   char szPath[_MAX_PATH];
   char szTemp[_MAX_PATH];
   char szCmd [_MAX_PATH];

   GetTempPath(sizeof(szPath), szPath);
   GetTempFileName(szPath, "vsa", 0, szTemp);

   _snprintf(szCmd, sizeof(szCmd), "cl /nologo /E %s > %s", szFile, szTemp);

   if(0 != system(szCmd))
      return D3DERR_INVALIDCALL;

   return D3DXAssembleShaderFromFile(szTemp, Flags, ppConstants, ppCode, ppErrors);
}
```

Note that for this function to work you must have Microsoft®Visual C++® installed and your environment set up so that Cl.exe is in your path.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

D3DXAssembleShaderFromFileW

D3DXAssembleShaderFromFileW

Assembles an ASCII description of a shader into binary form, where the source file name is a Unicode string.

```
HRESULT D3DXAssembleShaderFromFileW(
  LPCWSTR pSrcFile,
  DWORD Flags,
```

LPD3DXBUFFER* *ppConstants*,
LPD3DXBUFFER* *ppCompiledShader*,
LPD3DXBUFFER* *ppCompilationErrors*
);

Parameters

pSrcFile
[in] Pointer to the source file name, in Unicode format.

Flags
[in] A combination of the following flags, specifying assembly options.

D3DXASM_DEBUG
Inserts debugging information as comments in the assembled shader.

D3DXASM_SKIPVALIDATION
Do not validate the generated code against known capabilities and constraints. This option is recommended only when assembling a shader you know will function (that is, the shader has been assembled before without this option.)

ppConstants
[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned constant declarations. These constants are returned as a vertex shader declaration fragment. It is up to the application to insert the contents of this buffer into their declaration. For pixel shaders this parameter is meaningless because constant declarations are included in the assembled shader. This parameter is ignored if it is NULL.

ppCompiledShader
[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned compiled object code. This parameter is ignored if it is NULL.

ppCompilationErrors
[out] Returns a pointer to an **ID3DXBuffer** interface, representing the returned ASCII error messages. This parameter is ignored if it is NULL.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

D3DXAssemblePixelShaderFromFile maps to either **D3DXAssembleShaderFromFileA** or **D3DXAssembleShaderFromFileW**,

depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings.

The following code fragment shows how **D3DXAssembleShaderFromFile** is defined.

```
#ifdef UNICODE
#define D3DXAssembleShaderFromFile D3DXAssembleShaderFromFileW
#else
#define D3DXAssembleShaderFromFile D3DXAssembleShaderFromFileA
#endif
```

The following example wrapper function, **PreprocessAndAssembleShaderFromFile**, shows how you can invoke the C preprocessor on your vertex shader code before invoking the assembler with **D3DXAssembleShaderFromFile**.

```
HRESULT
    PreprocessAndAssembleShaderFromFile(
        LPCSTR          szFile,
        DWORD           Flags,
        LPD3DXBUFFER*   ppConstants,
        LPD3DXBUFFER*   ppCode,
        LPD3DXBUFFER*   ppErrors)
{
    char szPath[_MAX_PATH];
    char szTemp[_MAX_PATH];
    char szCmd [_MAX_PATH];

    GetTempPath(sizeof(szPath), szPath);
    GetTempFileName(szPath, "vsa", 0, szTemp);

    _snprintf(szCmd, sizeof(szCmd), "cl /nologo /E %s > %s", szFile, szTemp);

    if(0 != system(szCmd))
        return D3DERR_INVALIDCALL;

    return D3DXAssembleShaderFromFile(szTemp, Flags, ppConstants, ppCode, ppErrors);
}
```

Note that for this function to work you must have Microsoft®Visual C++® installed and your environment set up so that Cl.exe is in your path.

Requirements

Header: Declared in D3dx8core.h.
Import Library: Use D3dx8.lib.

See Also

D3DXAssembleShaderFromFileA

Shape-Drawing Functions

Direct3DX supplies the following shape-drawing functions.

- **D3DXCreateBox**
- **D3DXCreateCylinder**
- **D3DXCreatePolygon**
- **D3DXCreateSphere**
- **D3DXCreateTeapot**
- **D3DXCreateTextA**
- **D3DXCreateTextW**
- **D3DXCreateTorus**

All shapes are returned as mesh objects.

These functions can be used in either a left-handed or right-handed coordinate system.

D3DXCreateBox

Uses a left-handed coordinate system to create a mesh containing an axis-aligned box.

```
HRESULT D3DXCreateBox(
  LPDIRECT3DDEVICE8 pDevice,
  FLOAT Width,
  FLOAT Height,
  FLOAT Depth,
  LPD3DXMESH* ppMesh,
  LPD3DXBUFFER* ppAdjacency
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device associated with the created box mesh.

Width
[in] Width of the box, along the x-axis.

Height
[in] Height of the box, along the y-axis.

Depth
[in] Depth of the box, along the z-axis.

ppMesh
[out] Address of a pointer to the output shape, an **ID3DXMesh** interface.

ppAdjacency
[out] Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh. NULL can be specified.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

The created box is centered at the origin.

Requirements

Header: Declared in D3dx8shape.h.
Import Library: Use D3dx8.lib.

D3DXCreateCylinder

Uses a left-handed coordinate system to create a mesh containing a cylinder.

```
HRESULT D3DXCreateCylinder(
  LPDIRECT3DDEVICE8 pDevice,
  FLOAT Radius1,
  FLOAT Radius2,
  FLOAT Length,
  UINT Slices,
  UINT Stacks,
  LPD3DXMESH* ppMesh,
  LPD3DXBUFFER* ppAdjacency
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device associated with the created cylinder mesh.

Radius1
[in] Radius at the negative Z end. Value should be greater than or equal to 0.0f.

Radius2
[in] Radius at the positive Z end. Value should be greater than or equal to 0.0f.

Length
[in] Length of the cylinder along the z-axis.

Slices
[in] Number of slices about the main axis.

Stacks
[in] Number of stacks along the main axis.

ppMesh
[out] Address of a pointer to the output shape, an **ID3DXMesh** interface.

ppAdjacency
[out] Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh. NULL can be specified.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

The created cylinder is centered at the origin, and its axis is aligned with the z-axis.

Requirements

Header: Declared in D3dx8shape.h.
Import Library: Use D3dx8.lib.

D3DXCreatePolygon

Uses a left-handed coordinate system to create a mesh containing an *n*-sided polygon.

```
HRESULT D3DXCreatePolygon(
  LPDIRECT3DDEVICE8 pDevice,
  FLOAT Length,
  UINT Sides,
  LPD3DXMESH* ppMesh,
```

LPD3DXBUFFER* *ppAdjacency*
);

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device associated with the created polygon mesh.

Length
[in] Length of each side.

Sides
[in] Number of sides for the polygon. Value must be greater than or equal to 3.

ppMesh
[out] Address of a pointer to the output shape, an **ID3DXMesh** interface.

ppAdjacency
[out] Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh. NULL can be specified.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

The created polygon is centered at the origin.

Requirements

Header: Declared in D3dx8shape.h.
Import Library: Use D3dx8.lib.

D3DXCreateSphere

Uses a left-handed coordinate system to create a mesh containing a sphere.

HRESULT D3DXCreateSphere(
LPDIRECT3DDEVICE8 *pDevice*,
FLOAT *Radius*,

```
  UINT Slices,
  UINT Stacks,
  LPD3DXMESH* ppMesh,
  LPD3DXBUFFER* ppAdjacency
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device associated with the created sphere mesh.

Radius
[in] Radius of the sphere. This value should be greater than or equal to 0.0f.

Slices
[in] Number of slices about the main axis.

Stacks
[in] Number of stacks along the main axis.

ppMesh
[out] Address of a pointer to the output shape, an **ID3DXMesh** interface.

ppAdjacency
[out] Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh. NULL can be specified.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

The created sphere is centered at the origin, and its axis is aligned with the z-axis.

Requirements

Header: Declared in D3dx8shape.h.
Import Library: Use D3dx8.lib.

D3DXCreateTeapot

Uses a left-handed coordinate system to create a mesh containing a teapot.

```
HRESULT D3DXCreateTeapot(
  LPDIRECT3DDEVICE8 pDevice,
  LPD3DXMESH* ppMesh,
  LPD3DXBUFFER* ppAdjacency
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device associated with the created teapot mesh.

ppMesh
[out] Address of a pointer to the output shape, an **ID3DXMesh** interface.

ppAdjacency
[out] Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh. NULL can be specified.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Requirements

Header: Declared in D3dx8shape.h.
Import Library: Use D3dx8.lib.

D3DXCreateTextA

Creates a mesh containing the specified ANSI text using the font associated with the device context.

```
HRESULT D3DXCreateTextA(
  LPDIRECT3DDEVICE8 pDevice,
  HDC hDC,
  LPCSTR pText,
```

```
  FLOAT Deviation,
  FLOAT Extrusion,
  LPD3DXMESH* ppMesh,
  LPGLYPHMETRICSFLOAT pGlyphMetrics
);
```

Parameters

pDevice

[in] Pointer to an **IDirect3DDevice8** interface, representing the device associated with the created text mesh.

hDC

[in] Device context, containing the font for output. The font selected by the device context must be a TrueType font.

pText

[in] Pointer to an ANSI string, specifying the text to generate.

Deviation

[in] Maximum chordal deviation from true font outlines.

Extrusion

[in] Amount to extrude text in the negative Z direction.

ppMesh

[out] Address of a pointer to the output shape, an **ID3DXMesh** interface.

pGlyphMetrics

[out] Pointer to an array of **GLYPHMETRICSFLOAT** structures to receive the glyph metric data. Each **GLYPHMETRICSFLOAT** structure in the array contains information about the placement and orientation of the corresponding glyph in the string. The number of elements in the array should be equal to the number of characters in the string. Note that the origin in each structure is not relative to the entire string, but rather is relative to that character cell. To compute the entire bounding box, add the increment for each glyph while traversing the string.

If you are not concerned with the glyph sizes, you can pass NULL to *pGlyphMetrics*.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

D3DXCreateText maps to either **D3DXCreateTextA** or **D3DXCreateTextW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateText** is defined.

```
#ifdef UNICODE
#define D3DXCreateText D3DXCreateTextW
#else
#define D3DXCreateText D3DXCreateTextA
#endif
```

For more information on the **GLYPHMETRICSFLOAT** structure, see the Microsoft® Platform Software Development Kit (SDK).

Requirements

Header: Declared in D3dx8shape.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextW

D3DXCreateTextW

Creates a mesh containing the specified Unicode text using the font associated with the device context.

```
HRESULT D3DXCreateTextW(
  LPDIRECT3DDEVICE8 pDevice,
  HDC hDC,
  LPCSTR pText,
  FLOAT Deviation,
  FLOAT Extrusion,
  LPD3DXMESH* ppMesh,
  LPGLYPHMETRICSFLOAT pGlyphMetrics
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device associated with the created text mesh.

hDC

[in] Device context, containing the font for output. The font selected by the device context must be a TrueType font.

pText

[in] Pointer to a Unicode string, specifying the text to generate.

Deviation

[in] Maximum chordal deviation from true font outlines.

Extrusion

[in] Amount to extrude text in the negative Z direction.

ppMesh

[out] Address of a pointer to the output shape, an **ID3DXMesh** interface.

pGlyphMetrics

[out] Pointer to an array of **GLYPHMETRICSFLOAT** structures to receive the glyph metric data. Each **GLYPHMETRICSFLOAT** structure in the array contains information about the placement and orientation of the corresponding glyph in the string. The number of elements in the array should be equal to the number of characters in the string. Note that the origin in each structure is not relative to the entire string, but rather is relative to that character cell. To compute the entire bounding box, add the increment for each glyph while traversing the string.

If you are not concerned with the glyph sizes, you can pass NULL to *pGlyphMetrics*.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

D3DXCreateText maps to either **D3DXCreateTextA** or **D3DXCreateTextW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateText** is defined.

```
#ifdef UNICODE
#define D3DXCreateText D3DXCreateTextW
#else
#define D3DXCreateText D3DXCreateTextA
#endif
```

For more information on the **GLYPHMETRICSFLOAT** structure, see the Microsoft® Platform Software Development Kit (SDK).

Requirements

Header: Declared in D3dx8shape.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextA

D3DXCreateTorus

Uses a left-handed coordinate system to create a mesh containing a torus.

```
HRESULT D3DXCreateTorus(
  LPDIRECT3DDEVICE8 pDevice,
  FLOAT InnerRadius,
  FLOAT OuterRadius,
  UINT Sides,
  UINT Rings,
  LPD3DXMESH* ppMesh,
  LPD3DXBUFFER* ppAdjacency
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device associated with the created torus mesh.

InnerRadius
[in] Inner-radius of the torus. Value should be greater than or equal to 0.0f.

OuterRadius
[in] Outer-radius of the torus. Value should be greater than or equal to 0.0f.

Sides
[in] Number of sides in a cross-section. Value must be greater than or equal to 3.

Rings
[in] Number of rings making up the torus. Value must be greater than or equal to 3.

ppMesh
[out] Address of a pointer to the output shape, an **ID3DXMesh** interface.

ppAdjacency

[out] Address of a pointer to an **ID3DXBuffer** interface. When the method returns, this parameter is filled with an array of three **DWORD**s per face that specify the three neighbors for each face in the mesh. NULL can be specified.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

The **D3DXCreateTorus** function draws a doughnut, centered at (0, 0, 0) whose axis is aligned with the z-axis. The inner-radius of the torus is the radius of the cross-section (the minor-radius), and the outer-radius of the torus is the radius of the central hole.

This function returns a mesh that can be used later for drawing or manipulation by the application.

Requirements

Header: Declared in D3dx8shape.h.
Import Library: Use D3dx8.lib.

Texturing Functions

Direct3DX supplies the following texturing functions.

- **D3DXCheckCubeTextureRequirements**
- **D3DXCheckTextureRequirements**
- **D3DXCheckVolumeTextureRequirements**
- **D3DXCreateCubeTexture**
- **D3DXCreateCubeTextureFromFileA**
- **D3DXCreateCubeTextureFromFileExA**
- **D3DXCreateCubeTextureFromFileExW**
- **D3DXCreateCubeTextureFromFileInMemory**
- **D3DXCreateCubeTextureFromFileInMemoryEx**
- **D3DXCreateCubeTextureFromFileW**
- **D3DXCreateTexture**

- **D3DXCreateTextureFromFileA**
- **D3DXCreateTextureFromFileExA**
- **D3DXCreateTextureFromFileExW**
- **D3DXCreateTextureFromFileInMemory**
- **D3DXCreateTextureFromFileInMemoryEx**
- **D3DXCreateTextureFromFileW**
- **D3DXCreateTextureFromResourceA**
- **D3DXCreateTextureFromResourceExA**
- **D3DXCreateTextureFromResourceExW**
- **D3DXCreateTextureFromResourceW**
- **D3DXCreateVolumeTexture**
- **D3DXFilterCubeTexture**
- **D3DXFilterTexture**
- **D3DXFilterVolumeTexture**
- **D3DXLoadSurfaceFromFileA**
- **D3DXLoadSurfaceFromFileInMemory**
- **D3DXLoadSurfaceFromFileW**
- **D3DXLoadSurfaceFromMemory**
- **D3DXLoadSurfaceFromResourceA**
- **D3DXLoadSurfaceFromResourceW**
- **D3DXLoadSurfaceFromSurface**
- **D3DXLoadVolumeFromMemory**
- **D3DXLoadVolumeFromVolume**

D3DXCheckCubeTextureRequirements

Checks cube-texture-creation parameters.

```
HRESULT D3DXCheckCubeTextureRequirements(
  LPDIRECT3DDEVICE8 pDevice,
  UINT* pSize,
  UINT* pNumMipLevels,
  DWORD Usage,
  D3DFORMAT* pFormat,
  D3DPOOL Pool
);
```

Parameters

pDevice

[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the cube texture.

pSize
[in, out] Pointer to the requested width and height in pixels, or NULL. Returns the corrected size.

pNumMipLevels
[in, out] Pointer to the number of requested mipmap levels, or NULL. Returns the corrected number of mipmap levels.

Usage
[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

pFormat
[in, out] Pointer to a member of the **D3DFORMAT** enumerated type. Specifies the desired pixel format, or NULL. Returns the corrected format.

Pool
[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the texture should be placed.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_INVALIDCALL

Remarks

If parameters to this function are invalid, this function returns corrected parameters.

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **IDirect3DCubeTexture8::GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXCheckTextureRequirements

Checks texture-creation parameters.

```
HRESULT D3DXCheckTextureRequirements(
  LPDIRECT3DDEVICE8 pDevice,
  UINT* pWidth,
  UINT* pHeight,
  UINT* pNumMipLevels,
  DWORD Usage,
  D3DFORMAT* pFormat,
  D3DPOOL Pool
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

pWidth
[in, out] Pointer to the requested width in pixels, or NULL. Returns the corrected size.

pHeight
[in, out] Pointer to the requested height in pixels, or NULL. Returns the corrected size.

pNumMipLevels
[in, out] Pointer to number of requested mipmap levels, or NULL. Returns the corrected number of mipmap levels.

Usage
[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

pFormat
[in, out] Pointer to a member of the **D3DFORMAT** enumerated type. Specifies the desired pixel format, or NULL. Returns the corrected format.

Pool
[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the texture should be placed.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_INVALIDCALL

Remarks

If parameters to this function are invalid, this function returns corrected parameters.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXCheckVolumeTextureRequirements

Checks volume-texture-creation parameters.

```
HRESULT D3DXCheckVolumeTextureRequirements(
  LPDIRECT3DDEVICE8 pDevice,
  UINT* pWidth,
  UINT* pHeight,
  UINT* pDepth,
  UINT* pNumMipLevels,
  DWORD Usage,
  D3DFORMAT* pFormat,
  D3DPOOL Pool
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the volume texture.

pWidth
[in, out] Pointer to the requested width in pixels, or NULL. Returns the corrected size.

pHeight
[in, out] Pointer to the requested height in pixels, or NULL. Returns the corrected size.

pDepth
[in, out] Pointer to the requested depth in pixels, or NULL. Returns the corrected size.

pNumMipLevels

[in, out] Pointer to the number of requested mipmap levels, or NULL. Returns the corrected number of mipmap levels.

Usage

[in] Currently not used, set to 0.

pFormat

[in, out] Pointer to a member of the **D3DFORMAT** enumerated type. Specifies the desired pixel format, or NULL. Returns the corrected format.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the volume texture should be placed.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_INVALIDCALL

Remarks

If parameters to this function are invalid, this function returns corrected parameters.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXCreateCubeTexture

Creates an empty cube texture, adjusting the calling parameters as needed.

```
HRESULT D3DXCreateCubeTexture(
  LPDIRECT3DDEVICE8 pDevice,
  UINT Size,
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  LPDIRECT3DCUBETEXTURE8* ppCubeTexture
);
```

Parameters

pDevice

[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

Size

[in] Width and height of the cube texture, in pixels. For example, if the cube texture is an 8-pixel by 8-pixel cube, the value for this parameter should be 8.

MipLevels

[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage

[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

Format

[in] Member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the cube texture. The returned cube texture might have a different format from that specified by *Format*. Applications should check the format of the returned cube texture.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the cube texture should be placed.

ppCubeTexture

[out] Address of a pointer to an **IDirect3DCubeTexture8** interface, representing the created cube texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **IDirect3DCubeTexture8::GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

Internally, **D3DXCreateCubeTexture** uses **D3DXCheckCubeTextureRequirements** to adjust the calling parameters. Therefore, calls to **D3DXCreateCubeTexture** will often succeed where calls to **IDirect3Ddevice8::CreateCubeTexture** would fail.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXCreateCubeTextureFromFileA

Creates a cube texture from a file specified by an ANSI string.

D3DXCreateCubeTextureFromFile maps to either **D3DXCreateCubeTextureFromFileA** or **D3DXCreateCubeTextureFromFileW**, depending on the inclusion or exclusion of the **#define UNICODE** switch, see Remarks.

```
HRESULT D3DXCreateCubeTextureFromFileA(
  LPDIRECT3DDEVICE8 pDevice,
  LPCSTR pSrcFile,
  LPDIRECT3DCUBETEXTURE8* ppCubeTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the cube texture.

pSrcFile
[in] Pointer to an ANSI string that specifies the file from which to create the cube texture. See Remarks.

ppCubeTexture
[out] Address of a pointer to an **IDirect3DCubeTexture8** interface, representing the created cube texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateCubeTextureFromFile** is defined.

```
#ifdef UNICODE
#define D3DXCreateCubeTextureFromFile D3DXCreateCubeTextureFromFileW
#else
#define D3DXCreateCubeTextureFromFile D3DXCreateCubeTextureFromFileA
#endif
```

Note that a resource created with this function will be placed in the memory class denoted by D3DPOOL_MANAGED.

D3DXCreateCubeTextureFromFileA uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format.

Requirements

Header: Declared in D3dx8tex.h.

See Also

D3DXCreateCubeTextureFromFileW

D3DXCreateCubeTextureFromFileExA

Creates a cube texture from a file specified by an ANSI string. This is a more advanced function than **D3DXCreateCubeTextureFromFileA**.

D3DXCreateCubeTextureFromFileEx maps to either **D3DXCreateCubeTextureFromFileExA** or **D3DXCreateCubeTextureFromFileExW**, depending on the inclusion or exclusion of the **#define UNICODE** switch, see Remarks.

```
HRESULT D3DXCreateCubeTextureFromFileExA(
  LPDIRECT3DDEVICE8 pDevice,
  LPCSTR pSrcFile,
  UINT Size,
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
```

```
  DWORD Filter,
  DWORD MipFilter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo,
  PALETTEENTRY* pPalette,
  LPDIRECT3DCUBETEXTURE8* ppCubeTexture
);
```

Parameters

pDevice

[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the cube texture.

pSrcFile

[in] Pointer to an ANSI string that specifies the file from which to create the cube texture. See Remarks.

Size

[in] Width and height of the cube texture, in pixels. For example, if the cube texture is an 8-pixel by 8-pixel cube, the value for this parameter should be 8. If this value is 0 or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels

[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage

[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

Format

[in] Member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the cube texture. The returned cube texture might have a different format from that specified by *Format*. Applications should check the format of the returned cube texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the cube texture should be placed.

Filter

[in] A combination of one or more flags, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter. Internally, the flag D3DX_FILTER_MIRROR is always used.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags. This flag is always used internally for this function.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

MipFilter
[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX

Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter. Internally, the flag D3DX_FILTER_MIRROR is always used.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags. This flag is always used internally for this function.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant, and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

pPalette

[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL. See Remarks.

ppCubeTexture

[out] Address of a pointer to an **IDirect3DCubeTexture8** interface, representing the created cube texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **IDirect3DCubeTexture8::GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX 8.0®, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateCubeTextureFromFileEx** is defined.

```
#ifdef UNICODE
#define D3DXCreateCubeTextureFromFileEx D3DXCreateCubeTextureFromFileExW
#else
#define D3DXCreateCubeTextureFromFileEx D3DXCreateCubeTextureFromFileExA
#endif
```

D3DXCreateCubeTextureFromFileExA uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format.

Requirements

Header: Declared in D3dx8tex.h.

See Also

D3DXCreateCubeTextureFromFileExW

D3DXCreateCubeTextureFromFileExW

Creates a cube texture from a file specified by Unicode string. This is a more advanced function than **D3DXCreateCubeTextureFromFileW**.

D3DXCreateCubeTextureFromFileEx maps to either **D3DXCreateCubeTextureFromFileExA** or **D3DXCreateCubeTextureFromFileExW**, depending on the inclusion or exclusion of the **#define UNICODE** switch, see Remarks.

```
HRESULT D3DXCreateCubeTextureFromFileExW(
  LPDIRECT3DDEVICE8 pDevice,
  LPCWSTR pSrcFile,
  UINT Size,
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  DWORD Filter,
  DWORD MipFilter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo,
  PALETTEENTRY* pPalette,
  LPDIRECT3DCUBETEXTURE8* ppCubeTexture
);
```

Parameters

pDevice

[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the cube texture.

pSrcFile

[in] Pointer to a Unicode string that specifies the file from which to create the cube texture. See Remarks.

Size

[in] Width and height of the cube texture, in pixels. For example, if the cube texture is an 8-pixel by 8-pixel cube, the value for this parameter should be 8. If this value is 0 or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels

[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage

[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**. See Remarks.

Format

[in] Member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the cube texture. The returned cube texture might have a different format from that specified by *Format*. Applications should check the format of the returned cube texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the cube texture should be placed.

Filter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
: Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
: Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
: No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
: Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
: Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter. Internally, the flag D3DX_FILTER_MIRROR is always used.

D3DX_FILTER_MIRROR

Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags. This flag is always used internally for this function.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

MipFilter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter. Internally, the flag D3DX_FILTER_MIRROR is always used.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and

D3DX_FILTER_MIRROR_W flags. This flag is always used internally for this function.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

pPalette
[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL. See Remarks.

ppCubeTexture
[out] Address of a pointer to an **IDirect3DCubeTexture8** interface, representing the created cube texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **IDirect3DCubeTexture8::GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateCubeTextureFromFileEx** is defined.

```
#ifdef UNICODE
#define D3DXCreateCubeTextureFromFileEx D3DXCreateCubeTextureFromFileExW
#else
#define D3DXCreateCubeTextureFromFileEx D3DXCreateCubeTextureFromFileExA
#endif
```

D3DXCreateCubeTextureFromFileExW uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format.

Requirements

Header: Declared in D3dx8tex.h.

See Also

D3DXCreateCubeTextureFromFileExA

D3DXCreateCubeTextureFromFileInMemory

Creates a cube texture from a file in memory.

```
HRESULT D3DXCreateCubeTextureFromFileInMemory(
  LPDIRECT3DDEVICE8 pDevice,
  LPCVOID pSrcData,
  UINT SrcDataSize,
  LPDIRECT3DCUBETEXTURE8* ppCubeTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the cube texture.

pSrcData
[in] Pointer to the file in memory from which to create the cubemap. See Remarks.

SrcDataSize
[in] Size of the file in memory, in bytes.

ppCubeTexture
[out] Address of a pointer to an **IDirect3DCubeTexture8** interface, representing the created cube texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Note that a resource created with this function will be placed in the memory class denoted by D3DPOOL_MANAGED.

This method is designed to be used for loading image files stored as RT_RCDATA, which is an application-defined resource (raw data). Otherwise this method will fail.

D3DXCreateCubeTextureFromFileInMemory uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format.

Requirements

Header: Declared in D3dx8tex.h.

D3DXCreateCubeTextureFromFileInMemoryEx

Creates a cube texture from a file in memory. This is a more advanced function than **D3DXCreateCubeTextureFromFileInMemory**.

```
HRESULT D3DXCreateCubeTextureFromFileInMemoryEx(
  LPDIRECT3DDEVICE8 pDevice,
  LPCVOID pSrcData,
  UINT SrcDataSize,
  UINT Size,
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  DWORD Filter,
  DWORD MipFilter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo,
  PALETTEENTRY* pPalette,
  LPDIRECT3DCUBETEXTURE8* ppCubeTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the cube texture.

pSrcData
[in] Pointer to the file in memory from which to create the cube texture. See Remarks.

SrcDataSize
[in] Size of the file in memory, in bytes.

Size
[in] Width and height in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels
[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage
[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET

is specified, the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**. See Remarks.

Format

[in] Member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the cube texture. The returned texture might have a different format from that specified by *Format*. Applications should check the format of the returned texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the cube texture should be placed.

Filter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter. Internally, the flag D3DX_FILTER_MIRROR is always used.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags. This flag is always used internally for this function.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

MipFilter
[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter. Internally, the flag D3DX_FILTER_MIRROR is always used.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags. This flag is always used internally for this function.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant, and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

pPalette
[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL. See Remarks.

ppCubeTexture
[out] Address of a pointer to an **IDirect3DCubeTexture8** interface, representing the created cube texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **IDirect3DCubeTexture8::GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

This method is designed to be used for loading image files stored as RT_RCDATA, which is an application-defined resource (raw data). Otherwise this method will fail.

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in

the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DXCreateCubeTextureFromFileInMemoryEx uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format

Requirements

Header: Declared in D3dx8tex.h.

D3DXCreateCubeTextureFromFileW

Creates a cube texture from a file specified by a Unicode string.

D3DXCreateCubeTextureFromFile maps to either **D3DXCreateCubeTextureFromFileA** or **D3DXCreateCubeTextureFromFileW**, depending on the inclusion or exclusion of the **#define UNICODE** switch, see Remarks.

```
HRESULT D3DXCreateCubeTextureFromFileW(
  LPDIRECT3DDEVICE8 pDevice,
  LPCWSTR pSrcFile,
  LPDIRECT3DCUBETEXTURE8* ppCubeTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the cube texture.

pSrcFile
[in] Pointer to a Unicode string that specifies the file from which to create the cube texture. See Remarks.

ppCubeTexture
[out] Address of a pointer to an **IDirect3DCubeTexture8** interface, representing the created cube texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateCubeTextureFromFile** is defined.

```
#ifdef UNICODE
#define D3DXCreateCubeTextureFromFile D3DXCreateCubeTextureFromFileW
#else
#define D3DXCreateCubeTextureFromFile D3DXCreateCubeTextureFromFileA
#endif
```

Note that a resource created with this function will be placed in the memory class denoted by D3DPOOL_MANAGED.

D3DXCreateCubeTextureFromFileInMemoryW uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format.

Requirements

Header: Declared in D3dx8tex.h.

See Also

D3DXCreateCubeTextureFromFileA

D3DXCreateTexture

Creates an empty texture, adjusting the calling parameters as needed.

```
HRESULT D3DXCreateTexture(
  LPDIRECT3DDEVICE8 pDevice,
  UINT Width,
  UINT Height,
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

Width
[in] Width in pixels. This value must be non-zero.

Height
[in] Height in pixels. This value must be non-zero.

MipLevels
[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage
[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

Format
[in] Member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the texture. The returned texture might have a different format from that specified by *Format*. Applications should check the format of the returned texture.

Pool
[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the texture should be placed.

ppTexture
[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

Internally, **D3DXCreateTexture** uses **D3DXCheckTextureRequirements** to adjust the calling parameters. Therefore, calls to **D3DXCreateTexture** will often succeed where calls to **IDirect3Ddevice8::CreateTexture** would fail.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXCreateTextureFromFileA

Creates a texture from a file specified by an ANSI string.

```
HRESULT D3DXCreateTextureFromFileA(
  LPDIRECT3DDEVICE8 pDevice,
  LPCSTR pSrcFile,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

pSrcFile
[in] Pointer to an ANSI string that specifies the file from which to create the texture.

ppTexture
[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Mipmapped textures automatically have each level filled with the loaded texture.

When loading images into mipmapped textures, some devices are unable to go to a 1x1 image and this function will fail. If this happens, then the images need to be loaded manually.

D3DXCreateTextureFromFile maps to either **D3DXCreateTextureFromFileA** or **D3DXCreateTextureFromFileW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateTextureFromFile** is defined.

```
#ifdef UNICODE
#define D3DXCreateTextureFromFile D3DXCreateTextureFromFileW
#else
#define D3DXCreateTextureFromFile D3DXCreateTextureFromFileA
#endif
```

Note that a resource created with this function will be placed in the memory class denoted by D3DPOOL_MANAGED.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromFileExA, **D3DXCreateTextureFromFileExW**, **D3DXCreateTextureFromFileW**

D3DXCreateTextureFromFileExA

Creates a texture from a file specified by an ANSI string. This is a more advanced function than **D3DXCreateTextureFromFileA**.

HRESULT D3DXCreateTextureFromFileExA(
LPDIRECT3DDEVICE8 *pDevice*,
LPCSTR *pSrcFile*,
UINT *Width*,
UINT *Height*,
UINT *MipLevels*,
DWORD *Usage*,
D3DFORMAT *Format*,
D3DPOOL *Pool*,
DWORD *Filter*,
DWORD *MipFilter*,

```
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo,
  PALETTEENTRY* pPalette,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice

[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

pSrcFile

[in] Pointer to an ANSI string that specifies the file from which to create the texture.

Width

[in] Width in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

Height

[in] Height, in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels

[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage

[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, *Pool* must be set to D3DPOOL_DEFAULT, and the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

Format

Member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the texture. The returned texture might have a different format from that specified by *Format*. Applications should check the format of the returned texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the texture should be placed.

Filter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

MipFilter
[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR

Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled in with a description of the data in the source image file, or NULL.

pPalette
[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL. See Remarks.

ppTexture
[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Mipmapped textures automatically have each level filled with the loaded texture.

When loading images into mipmapped textures, some devices are unable to go to a 1x1 image and this function will fail. If this happens, then the images need to be loaded manually.

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DXCreateTextureFromFileEx maps to either **D3DXCreateTextureFromFileExA** or **D3DXCreateTextureFromFileExW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateTextureFromFileEx** is defined.

```
#ifdef UNICODE
#define D3DXCreateTextureFromFileEx D3DXCreateTextureFromFileExW
#else
#define D3DXCreateTextureFromFileEx D3DXCreateTextureFromFileExA
#endif
```

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromFileA, **D3DXCreateTextureFromFileExW**, **D3DXCreateTextureFromFileW**

D3DXCreateTextureFromFileExW

Creates a texture from a file specified by Unicode string. This is a more advanced function than **D3DXCreateTextureFromFileW**.

```
HRESULT D3DXCreateTextureFromFileExW(
  LPDIRECT3DDEVICE8 pDevice,
  LPCWSTR pSrcFile,
  UINT Width,
  UINT Height,
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  DWORD Filter,
  DWORD MipFilter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo,
  PALETTEENTRY* pPalette,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

pSrcFile
[in] Pointer to a Unicode string that specifies the file from which to create the texture.

Width
[in] Width in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

Height
[in] Height in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels
[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage

[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, *Pool* must be set to D3DPOOL_DEFAULT, and the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

Format

[in] Member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the texture. The returned texture might have a different format from that specified by *Format*. Applications should check the format of the returned texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the texture should be placed.

Filter

[in] A combination of one ore more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

MipFilter
[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V

Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

pPalette
[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL. See Remarks.

ppTexture
[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Mipmapped textures automatically have each level filled with the loaded texture.

When loading images into mipmapped textures, some devices are unable to go to a 1x1 image and this function will fail. If this happens, then the images need to be loaded manually.

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in

the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DXCreateTextureFromFileEx maps to either **D3DXCreateTextureFromFileExA** or **D3DXCreateTextureFromFileExW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateTextureFromFileEx** is defined.

```
#ifdef UNICODE
#define D3DXCreateTextureFromFileEx D3DXCreateTextureFromFileExW
#else
#define D3DXCreateTextureFromFileEx D3DXCreateTextureFromFileExA
#endif
```

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromFileA, **D3DXCreateTextureFromFileExA**, **D3DXCreateTextureFromFileW**

D3DXCreateTextureFromFileInMemory

Creates a texture from a file in memory.

```
HRESULT D3DXCreateTextureFromFileInMemory(
  LPDIRECT3DDEVICE8 pDevice,
  LPCVOID pSrcData,
  UINT SrcData,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

pSrcData
[in] Pointer to the file in memory from which to create the texture.

SrcData
[in] Size of the file in memory, in bytes.

ppTexture

[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Note that a resource created with this function will be placed in the memory class denoted by D3DPOOL_MANAGED.

This method is designed to be used for loading image files stored as RT_RCDATA, which is an application-defined resource (raw data). Otherwise this method will fail.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromFileInMemoryEx

D3DXCreateTextureFromFileInMemoryEx

Creates a texture from a file in memory. This is a more advanced function than **D3DXCreateTextureFromFileInMemory**.

HRESULT D3DXCreateTextureFromFileInMemoryEx(
LPDIRECT3DDEVICE8 *pDevice*,
LPCVOID *pSrcData*,
UINT *SrcData*,
UINT *Width*,
UINT *Height*,
UINT *MipLevels*,
DWORD *Usage*,
D3DFORMAT *Format*,

```
  D3DPOOL Pool,
  DWORD Filter,
  DWORD MipFilter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo
  PALETTEENTRY* pPalette,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

pSrcData
[in] Pointer to the file in memory from which to create the texture.

SrcData
[in] Size of the file in memory, in bytes.

Width
[in] Width in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

Height
[in] Height, in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels
[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage
[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, *Pool* must be set to D3DPOOL_DEFAULT, and the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

Format
[in] A member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the texture. The returned texture might have a different format from that specified by *Format*. Applications should check the format of the returned texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool
[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the texture should be placed.

Filter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
: Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
: Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
: No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
: Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
: Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
: Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
: Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
: Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
: Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
: The resulting image must be dithered using a 4x4 ordered dither algorithm.

MipFilter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

pPalette

[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL. See Remarks.

ppTexture

[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

This method is designed to be used for loading image files stored as RT_RCDATA, which is an application-defined resource (raw data). Otherwise, this method will fail.

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromFileInMemory

D3DXCreateTextureFromFileW

Creates a texture from a file specified by a Unicode string.

HRESULT D3DXCreateTextureFromFileW(
LPDIRECT3DDEVICE8 *pDevice*,
LPCWSTR *pSrcFile*,
LPDIRECT3DTEXTURE8* *ppTexture*

```
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

pSrcFile
[in] Pointer to a Unicode string that specifies the file from which to create the texture.

ppTexture
[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

Mipmapped textures automatically have each level filled with the loaded texture.

When loading images into mipmapped textures, some devices are unable to go to a 1x1 image and this function will fail. If this happens, then the images need to be loaded manually.

D3DXCreateTextureFromFile maps to either **D3DXCreateTextureFromFileA** or **D3DXCreateTextureFromFileW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateTextureFromFile** is defined.

```
#ifdef UNICODE
#define D3DXCreateTextureFromFile D3DXCreateTextureFromFileW
#else
#define D3DXCreateTextureFromFile D3DXCreateTextureFromFileA
#endif
```

Note that a resource created with this function will be placed in the memory class denoted by D3DPOOL_MANAGED.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromFileA, **D3DXCreateTextureFromFileExA**, **D3DXCreateTextureFromFileExW**

D3DXCreateTextureFromResourceA

Creates a texture from a resource specified by an ANSI string.

```
HRESULT D3DXCreateTextureFromResourceA(
  LPDIRECT3DDEVICE8 pDevice,
  HMODULE hSrcModule,
  LPCSTR pSrcResource,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

hSrcModule
[in] Handle to the module where the resource is located, or NULL for module associated with the image the operating system used to create the current process.

pSrcResource
[in] Pointer to an ANSI string that specifies the resource from which to create the texture.

ppTexture
[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

The resource being loaded must be a bitmap resource(RT_BITMAP).

D3DXCreateTextureFromResource maps to either **D3DXCreateTextureFromResourceA** or **D3DXCreateTextureFromResourceW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateTextureFromResource** is defined.

```
#ifdef UNICODE
#define D3DXCreateTextureFromResource D3DXCreateTextureFromResourceW
#else
#define D3DXCreateTextureFromResource D3DXCreateTextureFromResourceA
#endif
```

Note that a resource created with this function will be placed in the memory class denoted by D3DPOOL_MANAGED.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromResourceExA, **D3DXCreateTextureFromResourceExW**, **D3DXCreateTextureFromResourceW**

D3DXCreateTextureFromResourceExA

Creates a texture from a resource specified by an ANSI string. This is a more advanced function than **D3DXCreateTextureFromResourceA**.

HRESULT D3DXCreateTextureFromResourceExA(
LPDIRECT3DDEVICE8 *pDevice*,
HMODULE *hSrcModule*,
LPCSTR *pSrcResource*,
UINT *Width*,
UINT *Height*,

```
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  DWORD Filter,
  DWORD MipFilter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo,
  PALETTEENTRY* pPalette,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

hSrcModule
[in] Handle to the module where the resource is located, or NULL for module associated with the image the operating system used to create the current process.

pSrcResource
[in] Pointer to an ANSI string that specifies the resource from which to create the texture.

Width
[in] Width in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

Height
[in] Height, in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels
[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage
[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, *Pool* must be set to D3DPOOL_DEFAULT, and the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

Format
[in] A member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the texture. The returned texture might have a different format from that specified by *Format*. Applications should check the format of the

returned texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the texture should be placed.

Filter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

MipFilter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

D3DX_FILTER_BOX

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX

Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR

Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE

No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT

Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE

Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR

Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U

Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V

Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W

Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER

The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey

[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image

format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo

[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

pPalette

[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL. See Remarks.

ppTexture

[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

The resource being loaded must be a bitmap resource(RT_BITMAP).

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DXCreateTextureFromResourceEx maps to either **D3DXCreateTextureFromResourceExA** or **D3DXCreateTextureFromResourceExW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateTextureFromResourceEx** is defined.

```
#ifdef UNICODE
#define D3DXCreateTextureFromResourceEx D3DXCreateTextureFromResourceExW
#else
#define D3DXCreateTextureFromResourceEx D3DXCreateTextureFromResourceExA
#endif
```

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromResourceA, **D3DXCreateTextureFromResourceW**, **D3DXCreateTextureFromResourceExW**

D3DXCreateTextureFromResourceExW

Creates a texture from a resource specified by a Unicode string. This is a more advanced function than **D3DXCreateTextureFromResourceW**.

```
HRESULT D3DXCreateTextureFromResourceExW(
  LPDIRECT3DDEVICE8 pDevice,
  HMODULE hSrcModule,
  LPCWSTR pSrcResource,
  UINT Width,
  UINT Height,
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  DWORD Filter,
  DWORD MipFilter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo,
  PALETTEENTRY* pPalette,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

hSrcModule
[in] Handle to the module where the resource is located, or NULL for module associated with the image the operating system used to create the current process.

pSrcResource
[in] Pointer to a Unicode string that specifies the resource from which to create the texture.

Width

[in] Width in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

Height

[in] Height, in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels

[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage

[in] 0 or D3DUSAGE_RENDERTARGET. Setting this flag to D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can then be passed to the *pNewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, *Pool* must be set to D3DPOOL_DEFAULT, and the application should check that the device supports this operation by calling **IDirect3D8::CheckDeviceFormat**.

Format

[in] A member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the texture. The returned texture might have a different format from that specified by *Format*. Applications should check the format of the returned texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the texture should be placed.

Filter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX

Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR

Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE

No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT

Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

MipFilter
[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pPalette
[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL. See Remarks.

ppTexture
[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

The resource being loaded must be a bitmap resource(RT_BITMAP).

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DXCreateTextureFromResourceEx maps to either **D3DXCreateTextureFromResourceExA** or **D3DXCreateTextureFromResourceExW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateTextureFromResourceEx** is defined.

```
#ifdef UNICODE
#define D3DXCreateTextureFromResourceEx D3DXCreateTextureFromResourceExW
#else
#define D3DXCreateTextureFromResourceEx D3DXCreateTextureFromResourceExA
#endif
```

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromResourceA, **D3DXCreateTextureFromResourceExA**, **D3DXCreateTextureFromResourceW**

D3DXCreateTextureFromResourceW

Creates a texture from a resource specified by a Unicode string.

```
HRESULT D3DXCreateTextureFromResourceW(
  LPDIRECT3DDEVICE8 pDevice,
  HMODULE hSrcModule,
  LPCWSTR pSrcResource,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Parameters

pDevice

[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the texture.

hSrcModule

[in] Handle to the module where the resource is located, or NULL for module associated with the image the operating system used to create the current process.

pSrcResource

[in] Pointer to a Unicode string that specifies the resource from which to create the texture.

ppTexture

[out] Address of a pointer to an **IDirect3DTexture8** interface, representing the created texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

Remarks

The resource being loaded must be a bitmap resource(RT_BITMAP).

D3DXCreateTextureFromResource maps to either **D3DXCreateTextureFromResourceA** or **D3DXCreateTextureFromResourceW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXCreateTextureFromResource** is defined.

```
#ifdef UNICODE
#define D3DXCreateTextureFromResource D3DXCreateTextureFromResourceW
#else
#define D3DXCreateTextureFromResource D3DXCreateTextureFromResourceA
#endif
```

Note that a resource created with this function will be placed in the memory class denoted by D3DPOOL_MANAGED.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXCreateTextureFromResourceA, **D3DXCreateTextureFromResourceExA**, **D3DXCreateTextureFromResourceExW**

D3DXCreateVolumeTexture

Creates an empty volume texture, adjusting the calling parameters as needed.

```
HRESULT D3DXCreateVolumeTexture(
  LPDIRECT3DDEVICE8 pDevice,
  UINT Width,
  UINT Height,
  UINT Depth,
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  LPDIRECT3DVOLUMETEXTURE8* ppVolumeTexture
);
```

Parameters

pDevice
[in] Pointer to an **IDirect3DDevice8** interface, representing the device to be associated with the volume texture.

Width
[in] Width in pixels. This value must be non-zero.

Height
[in] Height in pixels. This value must be non-zero.

Depth
[in] Depth in pixels. This value must be non-zero.

MipLevels
[in] Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage
[in] Currently not used, set to 0.

Format
[in] Member of the **D3DFORMAT** enumerated type, describing the requested pixel format for the volume texture. The returned volume texture might have a

different format from that specified by *Format*. Applications should check the format of the returned volume texture.

Pool

[in] Member of the **D3DPOOL** enumerated type, describing the memory class into which the volume texture should be placed.

ppVolumeTexture

[out] Address of a pointer to an **IDirect3DVolumeTexture8** interface, representing the created volume texture object.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DERR_INVALIDCALL

E_OUTOFMEMORY

Remarks

Internally, **D3DXCreateVolumeTexture** uses **D3DXCheckVolumeTextureRequirements** to adjust the calling parameters. Therefore, calls to **D3DXCreateVolumeTexture** will often succeed where calls to **IDirect3Ddevice8::CreateVolumeTexture** would fail.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXFilterCubeTexture

Filters mipmap levels of a cube texture map.

```
HRESULT D3DXFilterCubeTexture(
  LPDIRECT3DCUBETEXTURE8 pCubeTexture,
  CONST PALETTEENTRY* pPalette,
  UINT SrcLevel,
  DWORD MipFilter
);
```

Parameters

pCubeTexture

[in] Pointer to an **IDirect3DCubeTexture8** interface, representing the cube texture object to filter.

pPalette

[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL. If a palette is not specified, the default Microsoft® Direct3D® palette (an all opaque white palette) is provided. See Remarks.

SrcLevel

[in] The level whose image is used to generate the subsequent levels. Specifying D3DX_DEFAULT for this parameter is equivalent to specifying 0.

MipFilter

[in] A combination of one or more flags controlling how the mipmap is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V

Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

For each side of the cube texture, a box-filter is recursively applied to each level to generate the next level.

Cube textures created in the default pool (D3DPOOL_DEFAULT) cannot be used with **D3DXFilterCubeTexture** because a lock operation is needed on the object. Note that locks are prohibited on textures in the default pool.

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXFilterTexture

Filters mipmap levels of a texture.

```
HRESULT D3DXFilterTexture(
  LPDIRECT3DTEXTURE8 pTexture,
  CONST PALETTEENTRY* pPalette,
  UINT SrcLevel,
  DWORD MipFilter
);
```

Parameters

pTexture

[in] Pointer to an **IDirect3DTexture8** interface, representing the texture object to filter.

pPalette

[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL for nonpalletized formats. If a palette is not specified, the default Microsoft® Direct3D® palette (an all opaque white palette) is provided. See Remarks.

SrcLevel

[in] The level whose image is used to generate the subsequent levels. Specifying D3DX_DEFAULT for this parameter is equivalent to specifying 0.

MipFilter

[in] A combination of one or more flags controlling how the mipmap is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
: Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
: Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
: No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
: Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
: Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
: Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U

Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
This flag indicates that the resulting image be dithered using a 4x4 ordered dither algorithm.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

A box-filter is recursively applied to each texture level to generate the next texture level.

Textures created in the default pool (D3DPOOL_DEFAULT) cannot be used with **D3DXFilterTexture** because a lock operation is needed on the object. Note that locks are prohibited on textures in the default pool.

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXFilterVolumeTexture

Filters mipmap levels of a volume texture.

HRESULT D3DXFilterVolumeTexture(
LPDIRECT3DVOLUMETEXTURE8 *pVolumeTexture*,

```
  CONST PALETTEENTRY* pPalette,
  UINT SrcLevel,
  DWORD MipFilter
);
```

Parameters

pVolumeTexture

[in] Pointer to an **IDirect3DVolumeTexture8** interface, representing the volume texture object to filter.

pPalette

[out] Pointer to a **PALETTEENTRY** structure, representing a 256-color palette to fill in, or NULL for nonpalletized formats. If a palette is not specified, the default Microsoft® Direct3D® palette (an all opaque white palette) is provided. See Remarks.

SrcLevel

[in] The level whose image is used to generate the subsequent levels. Specifying D3DX_DEFAULT for this parameter is equivalent to specifying 0.

MipFilter

[in] A combination of one or more flags controlling how the mipmap is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX | D3D_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
: Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
: Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
: No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
: Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
: Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR

Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be D3DERR_INVALIDCALL.

Remarks

A box-filter is recursively applied to each texture level to generate the next texture level.

Volume textures created in the default pool (D3DPOOL_DEFAULT) cannot be used with **D3DXFilterVolumeTexture** because a lock operation is needed on the object. Note that locks are prohibited on textures in the default pool.

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXLoadSurfaceFromFileA

Loads a surface from a file specified by an ANSI string.

HRESULT D3DXLoadSurfaceFromFileA(
LPDIRECT3DSURFACE8 *pDestSurface*,
CONST PALETTEENTRY* *pDestPalette*,

```
  CONST RECT* pDestRect,
  LPCSTR pSrcFile,
  CONST RECT* pSrcRect,
  DWORD Filter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo
);
```

Parameters

pDestSurface

[in] Pointer to an **IDirect3DSurface8** interface. Specifies the destination surface, which receives the image.

pDestPalette

[in] Pointer to a **PALETTEENTRY** structure, the destination palette of 256 colors or NULL. See Remarks.

pDestRect

[in] Pointer to a **RECT** structure. Specifies the destination rectangle. Set this parameter to NULL to specify the entire surface.

pSrcFile

[in] Pointer to an ANSI string that specifies the file name of the source image.

pSrcRect

[in] Pointer to a **RECT** structure. Specifies the source rectangle. Set this parameter to NULL to specify the entire image.

Filter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX

Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR

Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE

No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT

Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE

Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DXLoadSurfaceFromFile maps to either **D3DXLoadSurfaceFromFileA** or **D3DXLoadSurfaceFromFileW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXLoadSurfaceFromFile** is defined.

```
#ifdef UNICODE
#define D3DXLoadSurfaceFromFile D3DXLoadSurfaceFromFileW
#else
#define D3DXLoadSurfaceFromFile D3DXLoadSurfaceFromFileA
#endif
```

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXLoadSurfaceFromFileW

D3DXLoadSurfaceFromFileInMemory

Loads a surface from a file in memory.

```
HRESULT D3DXLoadSurfaceFromFileInMemory(
  LPDIRECT3DSURFACE8 pDestSurface,
  CONST PALETTEENTRY* pDestPalette,
  CONST RECT* pDestRect,
  LPCVOID pSrcData,
  UINT SrcData,
  CONST RECT* pSrcRect,
  DWORD Filter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo
);
```

Parameters

pDestSurface
[in] Pointer to an **IDirect3DSurface8** interface. Specifies the destination surface, which receives the image.

pDestPalette
[in] Pointer to a **PALETTEENTRY** structure, the destination palette of 256 colors or NULL. See Remarks.

pDestRect

[in] Pointer to a **RECT** structure. Specifies the destination rectangle. Set this parameter to NULL to specify the entire surface.

pSrcData

[in] Pointer to the file in memory from which to load the surface.

SrcData

[in] Size of the file in memory, in bytes.

pSrcRect

[in] Pointer to a **RECT** structure. Specifies the source rectangle. Set this parameter to NULL to specify the entire image.

Filter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
: Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
: Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
: No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
: Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
: Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
: Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
: Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
: Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W

Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXLoadSurfaceFromFileW

Loads a surface from a file specified by a Unicode string.

```
HRESULT D3DXLoadSurfaceFromFileW(
  LPDIRECT3DSURFACE8 pDestSurface,
  CONST PALETTEENTRY* pDestPalette,
  CONST RECT* pDestRect,
  LPCWSTR pSrcFile,
  CONST RECT* pSrcRect,
  DWORD Filter,
```

```
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo
);
```

Parameters

pDestSurface
[in] Pointer to an **IDirect3DSurface8** interface. Specifies the destination surface, which receives the image.

pDestPalette
[in] Pointer to a **PALETTEENTRY** structure, the destination palette of 256 colors or NULL. See Remarks.

pDestRect
[in] Pointer to a **RECT** structure. Specifies the destination rectangle. Set this parameter to NULL to specify the entire surface.

pSrcFile
[in] Pointer to a Unicode string that specifies the file name of the source image.

pSrcRect
[in] Pointer to a **RECT** structure. Specifies the source rectangle. Set this parameter to NULL to specify the entire image.

Filter
[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DXLoadSurfaceFromFile maps to either **D3DXLoadSurfaceFromFileA** or **D3DXLoadSurfaceFromFileW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXLoadSurfaceFromFile** is defined.

```
#ifdef UNICODE
#define D3DXLoadSurfaceFromFile D3DXLoadSurfaceFromFileW
#else
#define D3DXLoadSurfaceFromFile D3DXLoadSurfaceFromFileA
#endif
```

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXLoadSurfaceFromFileA

D3DXLoadSurfaceFromMemory

Loads a surface from memory.

```
HRESULT D3DXLoadSurfaceFromMemory(
  LPDIRECT3DSURFACE8 pDestSurface,
  CONST PALETTEENTRY* pDestPalette,
  CONST RECT* pDestRect,
  LPCVOID pSrcMemory,
  D3DFORMAT SrcFormat,
  UINT SrcPitch,
  CONST PALETTEENTRY* pSrcPalette,
  CONST RECT* pSrcRect,
  DWORD Filter,
  D3DCOLOR ColorKey
);
```

Parameters

pDestSurface
[in] Pointer to an **IDirect3DSurface8** interface. Specifies the destination surface, which receives the image.

pDestPalette
[in] Pointer to a **PALETTEENTRY** structure, the destination palette of 256 colors or NULL. See Remarks.

pDestRect
[in] Pointer to a **RECT** structure. Specifies the destination rectangle. Set this parameter to NULL to specify the entire surface.

pSrcMemory
[in] Pointer to the top-left corner of the source image in memory.

SrcFormat
[in] Member of the **D3DFORMAT** enumerated type, the pixel format of the source image.

SrcPitch
[in] Pitch of source image, in bytes. For DXT formats, this number should represent the width of one row of cells, in bytes.

pSrcPalette
[in] Pointer to a **PALETTEENTRY** structure, the source palette of 256 colors or NULL. See Remarks.

pSrcRect
[in] Pointer to a **RECT** structure. Specifies the dimensions of the source image in memory. This value cannot be NULL.

Filter
[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V

Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXLoadSurfaceFromResourceA

Loads a surface from a resource specified by an ANSI string.

```
HRESULT D3DXLoadSurfaceFromResourceA(
  LPDIRECT3DSURFACE8 pDestSurface,
  CONST PALETTEENTRY* pDestPalette,
  CONST RECT* pDestRect,
  HMODULE hSrcModule,
  LPCSTR pSrcResource,
  CONST RECT* pSrcRect,
  DWORD Filter,
```

```
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo
);
```

Parameters

pDestSurface
[in] Pointer to an **IDirect3DSurface8** interface. Specifies the destination surface, which receives the image.

pDestPalette
[in] Pointer to a **PALETTEENTRY** structure, the destination palette of 256 colors or NULL. See Remarks.

pDestRect
[in] Pointer to a **RECT** structure. Specifies the destination rectangle. Set this parameter to NULL to specify the entire surface.

hSrcModule
[in] Handle to the module where the resource is located, or NULL for module associated with the image the operating system used to create the current process.

pSrcResource
[in] Pointer to an ANSI string that specifies the resource name of the source image.

pSrcRect
[in] Pointer to a **RECT** structure. Specifies the source rectangle. Set this parameter to NULL to specify the entire surface.

Filter
[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE

Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

The resource being loaded must be a bitmap resource(RT_BITMAP).

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in

the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DXLoadSurfaceFromResource maps to either **D3DXLoadSurfaceFromResourceA** or **D3DXLoadSurfaceFromResourceW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXLoadSurfaceFromResource** is defined.

```
#ifdef UNICODE
#define D3DXLoadSurfaceFromResource D3DXLoadSurfaceFromResourceW
#else
#define D3DXLoadSurfaceFromResource D3DXLoadSurfaceFromResourceA
#endif
```

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXLoadSurfaceFromResourceW

D3DXLoadSurfaceFromResourceW

Loads a surface from a resource specified by a Unicode string.

```
HRESULT D3DXLoadSurfaceFromResourceW(
  LPDIRECT3DSURFACE8 pDestSurface,
  CONST PALETTEENTRY* pDestPalette,
  CONST RECT* pDestRect,
  HMODULE hSrcModule,
  LPCWSTR pSrcResource,
  CONST RECT* pSrcRect,
  DWORD Filter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo
);
```

Parameters

pDestSurface
[in] Pointer to an **IDirect3DSurface8** interface. Specifies the destination surface, which receives the image.

pDestPalette

[in] Pointer to a **PALETTEENTRY** structure, the destination palette of 256 colors or NULL. See Remarks.

pDestRect

[in] Pointer to a **RECT** structure. Specifies the destination rectangle. Set this parameter to NULL to specify the entire surface.

hSrcModule

[in] Handle to the module where the resource is located, or NULL for module associated with the image the operating system used to create the current process.

pSrcResource

[in] Pointer to a Unicode string that specifies the resource name of the source image.

pSrcRect

[in] Pointer to a **RECT** structure. Specifies the source rectangle. Set this parameter to NULL to specify the entire surface.

Filter

[in] A combination of one or more flags controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX

Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR

Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE

No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT

Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE

Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR

Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U

Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

pSrcInfo
[in, out] Pointer to a **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or NULL.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

The resource being loaded must be a bitmap resource(RT_BITMAP).

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DXLoadSurfaceFromResource maps to either **D3DXLoadSurfaceFromResourceA** or **D3DXLoadSurfaceFromResourceW**, depending on the inclusion or exclusion of the **#define UNICODE** switch. Include or exclude the #**define UNICODE** switch to specify whether your application expects Unicode or ANSI strings. The following code fragment shows how **D3DXLoadSurfaceFromResource** is defined.

```
#ifdef UNICODE
#define D3DXLoadSurfaceFromResource D3DXLoadSurfaceFromResourceW
```

```
#else
#define D3DXLoadSurfaceFromResource D3DXLoadSurfaceFromResourceA
#endif
```

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

See Also

D3DXLoadSurfaceFromResourceA

D3DXLoadSurfaceFromSurface

Loads a surface from another surface with color conversion.

```
HRESULT D3DXLoadSurfaceFromSurface(
  LPDIRECT3DSURFACE8 pDestSurface,
  CONST PALETTEENTRY* pDestPalette,
  CONST RECT* pDestRect,
  LPDIRECT3DSURFACE8 pSrcSurface,
  CONST PALETTEENTRY* pSrcPalette,
  CONST RECT* pSrcRect,
  DWORD Filter,
  D3DCOLOR ColorKey
);
```

Parameters

pDestSurface
[in] Pointer to an **IDirect3DSurface8** interface. Specifies the destination surface, which receives the image.

pDestPalette
[in] Pointer to a **PALETTEENTRY** structure, the destination palette of 256 colors or NULL. See Remarks.

pDestRect
[in] Pointer to a **RECT** structure. Specifies the destination rectangle. Set this parameter to NULL to specify the entire surface.

pSrcSurface
[in] Pointer to an **IDirect3DSurface8** interface, representing the source surface.

pSrcPalette
[in] Pointer to a **PALETTEENTRY** structure, the source palette of 256 colors or NULL. See Remarks.

pSrcRect

[in] Pointer to a **RECT** structure. Specifies the source rectangle. Set this parameter to NULL to specify the entire surface.

Filter

[in] A combination of one or more flags, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX
Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey

[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image

format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXLoadVolumeFromMemory

Loads a volume from memory.

```
HRESULT D3DXLoadVolumeFromMemory(
  LPDIRECT3DVOLUME8 pDestVolume,
  CONST PALETTEENTRY* pDestPalette,
  CONST D3DBOX* pDestBox,
  LPCVOID pSrcMemory,
  D3DFORMAT SrcFormat,
  UINT SrcRowPitch,
  UINT SrcSlicePitch,
  CONST PALETTEENTRY* pSrcPalette,
  CONST D3DBOX* pSrcBox,
  DWORD Filter,
  D3DCOLOR ColorKey
);
```

Parameters

pDestVolume

[in] Pointer to an **IDirect3DVolume8** interface. Specifies the destination volume, which receives the image.

pDestPalette

[in] Pointer to a **PALETTEENTRY** structure, the destination palette of 256 colors or NULL. See Remarks.

pDestBox

[in] Pointer to a **D3DBOX** structure. Specifies the destination box. Set this parameter to NULL to specify the entire volume.

pSrcMemory

[in] Pointer to the top-left corner of the source volume in memory.

SrcFormat

[in] Member of the **D3DFORMAT** enumerated type, the pixel format of the source volume.

SrcRowPitch

[in] Pitch of source image, in bytes. For DXT formats (compressed texture formats), this number should represent the size of one row of cells, in bytes.

SrcSlicePitch

[in] Pitch of source image, in bytes. For DXT formats (compressed texture formats), this number should represent the size of one slice of cells, in bytes.

pSrcPalette

[in] Pointer to a **PALETTEENTRY** structure, the source palette of 256 colors or NULL. See Remarks.

pSrcBox

[in] Pointer to a **D3DBOX** structure. Specifies the source box. NULL is not a valid value for this parameter.

Filter

[in] A combination of one or more flags, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX

Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR

Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE

No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT

Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

D3DXLoadVolumeFromVolume

Loads a volume from another volume.

```
HRESULT D3DXLoadVolumeFromVolume(
  LPDIRECT3DVOLUME8 pDestVolume,
  CONST PALETTEENTRY* pDestPalette,
  CONST D3DBOX* pDestBox,
  LPDIRECT3DVOLUME8 pSrcVolume,
  CONST PALETTEENTRY* pSrcPalette,
  CONST D3DBOX* pSrcBox,
  DWORD Filter,
  D3DCOLOR ColorKey
);
```

Parameters

pDestVolume
[in] Pointer to an **IDirect3DVolume8** interface. Specifies the destination volume, which receives the image.

pDestPalette
[in] Pointer to a **PALETTEENTRY** structure, the destination palette of 256 colors or NULL. See Remarks.

pDestBox
[in] Pointer to a **D3DBOX** structure. Specifies the destination box. Set this parameter to NULL to specify the entire volume.

pSrcVolume
[in] A Pointer to an **IDirect3DVolume8** interface. Specifies the source volume.

pSrcPalette
[in] Pointer to a **PALETTEENTRY** structure, the source palette of 256 colors or NULL. See Remarks.

pSrcBox
[in] Pointer to a **D3DBOX** structure. Specifies the source box. Set this parameter to NULL to specify the entire volume.

Filter
[in] A combination of one or more flags, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_TRIANGLE | D3DX_FILTER_DITHER.

Each valid filter must contain exactly one of the following flags.

D3DX_FILTER_BOX

Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_LINEAR
Each destination pixel is computed by sampling the four nearest pixels from the source image. This filter works best when the scale on both axes is less than two.

D3DX_FILTER_NONE
No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_TRIANGLE
Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

D3DX_FILTER_MIRROR
Specifying this flag is the same as specifying the D3DX_FILTER_MIRROR_U, D3DX_FILTER_MIRROR_V, and D3DX_FILTER_MIRROR_W flags.

D3DX_FILTER_MIRROR_U
Pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
Pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_W
Pixels off the edge of the texture on the w-axis should be mirrored, not wrapped.

D3DX_FILTER_DITHER
The resulting image must be dithered using a 4x4 ordered dither algorithm.

ColorKey
[in] **D3DCOLOR** value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to 0xFF000000.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

Remarks

For details on **PALETTEENTRY**, see the Microsoft® Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALLETTEENTRY** structure does not function as documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Requirements

Header: Declared in D3dx8tex.h.
Import Library: Use D3dx8.lib.

Macros

This section contains reference information for the macros provided by the Direct3DX utility library.

- **D3DXToDegree**
- **D3DXToRadian**

D3DXToDegree

Converts radians into degrees.

```
D3DXToDegree( radian ) ((radian) * (180.0f / D3DX_PI))
```

Parameters

radian
The value, in radians, to convert to degrees.

Return Values

The macro returns the radian value in degrees.

Remarks

The CONSTant D3DX_PI is declared with the following define.

```
#define D3DX_PI    ((FLOAT)  3.141592654f)
```

Requirements

Header: Declared in D3dx8math.h.

See Also

D3DXToRadian

D3DXToRadian

Converts degrees into radians.

```
D3DXToRadian( degree ) ((degree) * (D3DX_PI / 180.0f))
```

Parameters

degree
The value, in degrees, to convert to radians.

Return Values

The macro returns the degree value in radians.

Remarks

The CONSTant D3DX_PI is declared with the following define.

```
#define D3DX_PI    ((FLOAT)  3.141592654f)
```

Requirements

Header: Declared in D3dx8math.H.

See Also

D3DXToDegree

Structures

This section contains information about the structures included in the Direct3DX utility library.

- **D3DXATTRIBUTERANGE**
- **D3DXATTRIBUTEWEIGHTS**
- **D3DXBONECOMBINATION**

- **D3DXCOLOR**
- **D3DXEFFECT_DESC**
- **D3DXIMAGE_INFO**
- **D3DXMATERIAL**
- **D3DXMATRIX**
- **D3DXPASS_DESC**
- **D3DXPARAMETER_DESC**
- **D3DXPLANE**
- **D3DXQUATERNION**
- **D3DXTECHNIQUE_DESC**
- **D3DXVECTOR2**
- **D3DXVECTOR3**
- **D3DXVECTOR4**
- **D3DXRTS_DESC**

D3DXATTRIBUTERANGE

Stores an attribute table entry.

```
typedef struct _D3DXATTRIBUTERANGE {
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount;
} D3DXATTRIBUTERANGE;
```

Members

AttribId
Attribute table identifier.

FaceStart
Starting face.

FaceCount
Face count.

VertexStart
Starting vertex.

VertexCount
Vertex count.

Remarks

An attribute table is used to identify areas of the mesh that need to be drawn with different textures, render states, materials, and so on. In addition, the application can use the attribute table to hide portions of a mesh by not drawing a given attribute identifier (**AttribId**) when drawing the frame.

The **LPD3DXATTRIBUTERANGE** type is defined as a pointer to the **D3DXATTRIBUTERANGE** structure.

```
typedef D3DXATTRIBUTERANGE* LPD3DXATTRIBUTERANGE;
```

Requirements

Header: Declared in D3dx8mesh.h.

See Also

ID3DXBaseMesh::DrawSubset

D3DXATTRIBUTEWEIGHTS

Specifies how to weigh vertex components.

```
typedef struct _D3DXATTRIBUTEWEIGHTS {
    FLOAT Position;
    FLOAT Boundary;
    FLOAT Normal;
    FLOAT Diffuse;
    FLOAT Specular;
    FLOAT Tex[8];
} D3DXATTRIBUTEWEIGHTS;
```

Members

Position
Position component weight.

Boundary
Boundary component weight.

Normal
Normal component weight.

Diffuse
Diffuse component weight.

Specular
Specular component weight.

Tex

Texture coordinate weights.

Remarks

This structure describes how a simplification operation will consider vertex data when calculating relative costs between collapsing edges. For example, if the **Normal** field is 0.0, then the simplification operation will ignore the vertex normal component when calculating the error for the collapse. However, if the **Normal** field is 1.0, then the simplification operation will use the vertex normal component. if the **Normal** field is 2.0, then double the amount of errors; if the Normal field is 4.0, then quadruple the number of errors, and so on.

The **LPD3DXATTRIBUTEWEIGHTS** type is defined as a pointer to the **D3DXATTRIBUTEWEIGHTS** structure.

```
typedef D3DXATTRIBUTEWEIGHTS* LPD3DXATTRIBUTEWEIGHTS;
```

Requirements

Header: Declared in D3dx8mesh.h.

See Also

D3DXCreateSPMesh, **D3DXGeneratePMesh**, **D3DXSimplifyMesh**

D3DXBONECOMBINATION

Describes a subset of the mesh that has the same attribute and bone combination.

```
typedef struct _D3DXBONECOMBINATION {
    DWORD AttribId;
    DWORD FaceStart;
    DWORD FaceCount;
    DWORD VertexStart;
    DWORD VertexCount;
    DWORD* BoneId;
} D3DXBONECOMBINATION, *LPD3DXBONECOMBINATION;
```

Members

AttribId
Attribute table identifier.

FaceStart
Starting face.

FaceCount
Face count.

VertexStart
Starting vertex.

VertexCount
Vertex count.

BoneId
Pointer to an array of values that identify each of the bones that can be drawn in a single drawing call. Note that the array can be of variable length to accommodate variable length bone combinations of **ID3DXSkinMesh::ConvertToIndexedBlendedMesh**.

Remarks

The subset of the mesh described by **D3DXBONECOMBINATION** can be rendered in a single drawing call.

Requirements

Header: Declared in D3dx8mesh.h.

D3DXCOLOR

Describes color values.

```
typedef struct D3DXCOLOR {
  FLOAT r, g, b, a;
} D3DXCOLOR;
```

Members

r
Red component of the color.

g
Green component of the color.

b
Blue component of the color.

a
Alpha component of the color.

Remarks

C++ programmers can take advantage of operator overloading and type casting. The C++ implementation of the **D3DXCOLOR** structure implements overloaded constructors and overloaded assignment, unary, and binary (including equality) operators. For more information, see C++ Specific Features.

Requirements

Header: Declared in D3dx8math.h.

D3DXEFFECT_DESC

Describes an effect object.

```
typedef struct _D3DXEFFECT_DESC
{
    UINT  Parameters;
    UINT  Techniques;
    DWORD Usage;

} D3DXEFFECT_DESC;
```

Members

Parameters
Number of parameters used for effect.

Techniques
Number of techniques that can render the effect.

Usage
Usage control for this effect. This member can have the following flag set:

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the effect will use software processing.

Remarks

An effect object can contain multiple rendering techniques and parameters for the same effect.

Requirements

Header: Declared in D3dx8effect.h.

See Also

ID3DXEffect::GetDesc

D3DXIMAGE_INFO

Returns a description of the original contents of an image file.

```
typedef struct _D3DXIMAGE_INFO {
    UINT Width;
    UINT Height;
```

```
    UINT Depth;
    UINT MipLevels;
    D3DFORMAT Format;
} D3DXIMAGE_INFO;
```

Members

Width
Width of original image in pixels.

Height
Height of original image in pixels.

Depth
Depth of original image in pixels.

MipLevels
Number of mip levels in original image.

Format
A value from the **D3DFORMAT** enumerated type that most closely describes the data in the original image.

Requirements

Header: Declared in D3dx8tex.h.

D3DXMATERIAL

Returns material information saved in Microsoft® DirectX® (.x) files.

```
struct D3DXMATERIAL {
    D3DMATERIAL8  MatD3D;
    LPSTR         pTextureFilename;
};
```

Members

MatD3D
D3DMATERIAL8 structure that describes the material properties.

pTextureFilename
Pointer to a string that specifies the file name of the texture.

Remarks

The **D3DXLoadMeshFromX** and **D3DXLoadMeshFromXof** functions return an array of **D3DXMATERIAL** structures that specify the material color and name of the texture for each material in the mesh. The application is then required to load the texture.

The **LPD3DXMATERIAL** type is defined as a pointer to the **D3DXMATERIAL** structure.

```
typedef struct D3DXMATERIAL *LPD3DXMATERIAL;
```

Requirements

Header: Declared in D3dx8mesh.h.

See Also

D3DXLoadMeshFromX, **D3DXLoadMeshFromXof**

D3DXMATRIX

Describes a matrix.

```
typedef struct D3DXMATRIX : public D3DMATRIX {
public:
    D3DXMATRIX() {};
    D3DXMATRIX( CONST FLOAT * );
    D3DXMATRIX( CONST D3DMATRIX& );
    D3DXMATRIX( FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44 );


    // access grants
    FLOAT& operator () ( UINT Row, UINT Col );
    FLOAT  operator () ( UINT Row, UINT Col ) const;

    // casting operators
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXMATRIX& operator *= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator += ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator -= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator *= ( FLOAT );
    D3DXMATRIX& operator /= ( FLOAT );

    // unary operators
    D3DXMATRIX operator + () const;
    D3DXMATRIX operator - () const;
```

```
    // binary operators
    D3DXMATRIX operator * ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator + ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator - ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator * ( FLOAT ) const;
    D3DXMATRIX operator / ( FLOAT ) const;

    friend D3DXMATRIX operator * ( FLOAT, CONST D3DXMATRIX& );

    BOOL operator == ( CONST D3DXMATRIX& ) const;
    BOOL operator != ( CONST D3DXMATRIX& ) const;

} D3DXMATRIX, *LPD3DXMATRIX;
```

Remarks

This structure inherits members from the **D3DMATRIX** structure. C programmers cannot use the **D3DXMATRIX** structure. They must use the **D3DMATRIX** structure.

In Direct3DX, the **_34** element of a projection matrix cannot be a negative number. If your application needs to use a negative value in this location, it should scale the entire projection matrix by -1, instead.

C++ programmers can take advantage of operator overloading and type casting. The C++ implementation of the **D3DXMATRIX** structure implements overloaded constructors and overloaded assignment, unary, and binary (including equality) operators. For more information, see C++ Specific Features.

Requirements

Header: Declared in D3dx8math.h.

D3DXPARAMETER_DESC

Describes a parameter used for an effect object.

```
typedef struct _D3DXPARAMETER_DESC
{
    DWORD           Name;
    D3DXPARAMETERTYPE Type;

} D3DXPARAMETER_DESC;
```

Members

Name
Four-character code that is the name of parameter.

Type

Describes the type of variable used for this parameter. This member is one of the following constants:

D3DXPT_DWORD
Parameter is a **DWORD** value.

D3DXPT_FLOAT
Parameter is a floating-point value.

D3DXPT_VECTOR
Parameter is a vector.

D3DXPT_MATRIX
Parameter is a 4x4 matrix.

D3DXPT_TEXTURE
Parameter is a texture.

D3DXPT_VERTEXSHADER
Parameter is a vertex shader.

D3DXPT_PIXELSHADER
Parameter is a pixel shader.

D3DXPT_CONSTANT
Parameter is a constant value.

D3DXPT_FORCE_DWORD
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.

See Also

ID3DXEffect::GetParameterDesc

D3DXPASS_DESC

Describes a pass for an effect object.

```
typedef struct _D3DXPASS_DESC
{
    DWORD Name;

} D3DXPASS_DESC;
```

Members

Name
Four-character code used for the name of the pass.

Remarks

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.

See Also

ID3DXTechnique::GetPassDesc

D3DXPLANE

Describes a plane.

```
typedef struct D3DXPLANE {
  FLOAT a, b, c, d;
} D3DXPLANE;
```

Members

a, **b**, **c**, and **d**
The a, b, c, and d coefficients, respectively, of the clipping plane in the general plane equation. See Remarks.

Remarks

The members of the **D3DXPLANE** structure take the form of the general plane equation. They fit into the general plane equation so that $ax + by + cz + dw = 0$.

C++ programmers can take advantage of operator overloading and type casting. The C++ implementation of the **D3DXPLANE** structure implements overloaded constructors and overloaded unary and binary operators. For more information, see C++ Specific Features.

Requirements

Header: Declared in D3dx8math.h.

D3DXQUATERNION

Describes a quaternion.

```
typedef struct D3DXQUATERNION {
  FLOAT x, y, z, w;
} D3DXQUATERNION;
```

Members

x
The x component.

y
The y component.

z
The z component.

w
The w component.

Remarks

Quaternions add a fourth element to the [*x, y, z*] values that define a vector, resulting in arbitrary 4-D vectors. However, the following illustrates how each element of a unit quaternion relates to an axis-angle rotation (where *q* represents a unit quaternion (x, y, z, w), *axis* is normalized, and *theta* is the desired CCW rotation about the axis):

```
q.x = sin(theta/2) * axis.x
q.y = sin(theta/2) * axis.y
q.z = sin(theta/2) * axis.z
q.w = cos(theta/2)
```

C++ programmers can take advantage of operator overloading and type casting. The C++ implementation of the **D3DXQUATERNION** structure implements overloaded constructors and overloaded assignment, unary and binary (including equality) operators. For more information, see C++ Specific Features.

Requirements

Header: Declared in D3dx8math.h.

D3DXTECHNIQUE_DESC

Describes a technique object.

```
typedef struct _D3DXTECHNIQUE_DESC
{
    DWORD Name;
```

```
    UINT  Passes;
} D3DXTECHNIQUE_DESC;
```

Members

Name
Four-character code used for the name of this technique object.

Passes
Number of rendering passes the technique requires. See Remarks.

Remarks

Some video cards can render two textures in a single pass. However, if a card does not have this capability, it is often possible to render the same effect in two passes using one texture for each pass.

A four-character code (FOURCC) is a string that is four characters in length. For more information, see **Four-Character Codes (FOURCC)**.

Requirements

Header: Declared in D3dx8effect.h.

See Also

ID3DXTechnique::GetDesc

D3DXVECTOR2

Describes a vector in two-dimensional (2-D) space.

```
typedef struct D3DXVECTOR2 {
  FLOAT x, y;
} D3DXVECTOR2;
```

Members

x
The x component.

y
The y component.

Remarks

C++ programmers can take advantage of operator overloading and type casting. The C++ implementation of the **D3DXVECTOR2** structure implements overloaded constructors and overloaded assignment, unary and binary (including equality) operators. For more information, see C++ Specific Features.

Requirements

Header: Declared in D3dx8math.h.

D3DXVECTOR3

Describes a vector in three-dimensional (3-D) space.

```
typedef struct D3DXVECTOR3 : public D3DVECTOR {
public:
    D3DXVECTOR3() {};
    D3DXVECTOR3( CONST FLOAT * );
    D3DXVECTOR3( CONST D3DVECTOR& );
    D3DXVECTOR3( FLOAT x, FLOAT y, FLOAT z );

    // casting
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXVECTOR3& operator += ( CONST D3DXVECTOR3& );
    D3DXVECTOR3& operator -= ( CONST D3DXVECTOR3& );
    D3DXVECTOR3& operator *= ( FLOAT );
    D3DXVECTOR3& operator /= ( FLOAT );

    // unary operators
    D3DXVECTOR3 operator + () const;
    D3DXVECTOR3 operator - () const;

    // binary operators
    D3DXVECTOR3 operator + ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator - ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator * ( FLOAT ) const;
    D3DXVECTOR3 operator / ( FLOAT ) const;

    friend D3DXVECTOR3 operator * ( FLOAT, CONST struct
D3DXVECTOR3& );

    BOOL operator == ( CONST D3DXVECTOR3& ) const;
    BOOL operator != ( CONST D3DXVECTOR3& ) const;

} D3DXVECTOR3, *LPD3DXVECTOR3;
```

Members

x

The x component.

y
The y component.

z
The z component.

Remarks

This structure inherits the **x**, **y** and **z** members from the **D3DVECTOR** structure. C programmers cannot use the **D3DXVECTOR3** structure. They must use the **D3DVECTOR** structure.

C++ programmers can take advantage of operator overloading and type casting. The C++ implementation of the **D3DXVECTOR3** structure implements overloaded constructors and overloaded assignment, unary and binary (including equality) operators. For more information, see C++ Specific Features.

Requirements

Header: Declared in D3dx8math.h.

D3DXVECTOR4

Describes a vector in four-dimensional (4-D) space.

```
typedef struct D3DXVECTOR4 {
  FLOAT x, y, z, w;
} D3DXVECTOR4;
```

Members

x
The x component.

y
The y component.

z
The z component.

w
The w component.

Remarks

C++ programmers can take advantage of operator overloading and type casting. The C++ implementation of the **D3DXVECTOR4** structure implements overloaded constructors and overloaded assignment, unary and binary (including equality) operators. For more information, see C++ Specific Features.

Requirements

Header: Declared in D3dx8math.h.

D3DXRTS_DESC

Describes a render surface.

```
typedef struct _D3DXRTS_DESC {
    UINT            Width;
    UINT            Height;
    D3DFORMAT       Format;
    BOOL            DepthStencil;
    D3DFORMAT       DepthStencilFormat;

} D3DXRTS_DESC;
```

Members

Width
Width of the render surface, in pixels.

Height
Height of the render surface, in pixels.

Format
Member of the **D3DFORMAT** enumerated type, describing the pixel format of the render surface.

DepthStencil
If TRUE, the render surface supports a depth-stencil surface; otherwise this member is set to FALSE.

DepthStencilFormat
If **DepthStencil** is set to TRUE, this parameter is a member of the **D3DFORMAT** enumerated type, describing the depth-stencil format of the render surface.

Requirements

Header: Declared in D3dx8core.h.

See Also

ID3DXRenderToSurface::GetDesc

C++ Specific Features

C++ programmers can take advantage of the constructor overloading, operator overloading, and type casting extensions offered by the Direct3DX utility library. The extensions are implemented in the following structures.

- **D3DXCOLOR** Extensions
- **D3DXMATRIX** Extensions
- **D3DXPLANE** Extensions
- **D3DXQUATERNION** Extensions
- **D3DXVECTOR2** Extensions
- **D3DXVECTOR3** Extensions
- **D3DXVECTOR4** Extensions

D3DXCOLOR Extensions

Supplies the following operator overloads and type casts for **D3DXCOLOR** structures.

```
#ifdef __cplusplus
public:
    D3DXCOLOR() {}
    D3DXCOLOR( DWORD argb );
    D3DXCOLOR( CONST FLOAT * );
    D3DXCOLOR( CONST D3DCOLORVALUE& );
    D3DXCOLOR( FLOAT r, FLOAT g, FLOAT b, FLOAT a );

    // casting
    operator DWORD () const;

    operator FLOAT* ();
    operator CONST FLOAT* () const;

    operator D3DCOLORVALUE* ();
    operator CONST D3DCOLORVALUE* () const;

    operator D3DCOLORVALUE& ();
    operator CONST D3DCOLORVALUE& () const;

    // assignment operators
    D3DXCOLOR& operator += ( CONST D3DXCOLOR& );
    D3DXCOLOR& operator -= ( CONST D3DXCOLOR& );
    D3DXCOLOR& operator *= ( FLOAT );
    D3DXCOLOR& operator /= ( FLOAT );
```

```
    // unary operators
    D3DXCOLOR operator + () const;
    D3DXCOLOR operator - () const;

    // binary operators
    D3DXCOLOR operator + ( CONST D3DXCOLOR& ) const;
    D3DXCOLOR operator - ( CONST D3DXCOLOR& ) const;
    D3DXCOLOR operator * ( FLOAT ) const;
    D3DXCOLOR operator / ( FLOAT ) const;

    friend D3DXCOLOR operator * (FLOAT, CONST D3DXCOLOR& );

    BOOL operator == ( CONST D3DXCOLOR& ) const;
    BOOL operator != ( CONST D3DXCOLOR& ) const;

#endif //__cplusplus
```

Requirements

Header: Declared in D3dx8mesh.h.

D3DXMATRIX Extensions

Supplies the following operator overloads and type casts for **D3DXMATRIX** structures.

```
#ifdef __cplusplus
typedef struct D3DXMATRIX : public D3DMATRIX
{
public:
    D3DXMATRIX() {};
    D3DXMATRIX( CONST FLOAT * );
    D3DXMATRIX( CONST D3DMATRIX& );
    D3DXMATRIX( FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44 );


    // access grants
    FLOAT& operator () ( UINT Row, UINT Col );
    FLOAT  operator () ( UINT Row, UINT Col ) const;

    // casting operators
    operator FLOAT* ();
```

```
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXMATRIX& operator *= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator += ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator -= ( CONST D3DXMATRIX& );
    D3DXMATRIX& operator *= ( FLOAT );
    D3DXMATRIX& operator /= ( FLOAT );

    // unary operators
    D3DXMATRIX operator + () const;
    D3DXMATRIX operator - () const;

    // binary operators
    D3DXMATRIX operator * ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator + ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator - ( CONST D3DXMATRIX& ) const;
    D3DXMATRIX operator * ( FLOAT ) const;
    D3DXMATRIX operator / ( FLOAT ) const;

    friend D3DXMATRIX operator * ( FLOAT, CONST D3DXMATRIX& );

    BOOL operator == ( CONST D3DXMATRIX& ) const;
    BOOL operator != ( CONST D3DXMATRIX& ) const;

} D3DXMATRIX, *LPD3DXMATRIX;

#else //!__cplusplus
```

Requirements

Header: Declared in D3dx8mesh.h.

D3DXPLANE Extensions

Supplies the following operator overloads and type casts for **D3DXPLANE** structures.

```
#ifdef __cplusplus
public:
    D3DXPLANE() {}
    D3DXPLANE( CONST FLOAT* );
    D3DXPLANE( FLOAT a, FLOAT b, FLOAT c, FLOAT d );

    // casting
    operator FLOAT* ();
```

```
    operator CONST FLOAT* () const;

    // unary operators
    D3DXPLANE operator + () const;
    D3DXPLANE operator - () const;

    // binary operators
    BOOL operator == ( CONST D3DXPLANE& ) const;
    BOOL operator != ( CONST D3DXPLANE& ) const;

#endif //__cplusplus
```

Requirements

Header: Declared in D3dx8mesh.h.

D3DXQUATERNION Extensions

Supplies the following operator overloads and type casts for **D3DXQUATERNION** structures.

```
#ifdef __cplusplus
public:
    D3DXQUATERNION() {}
    D3DXQUATERNION( CONST FLOAT * );
    D3DXQUATERNION( FLOAT x, FLOAT y, FLOAT z, FLOAT w );

    // casting
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXQUATERNION& operator += ( CONST D3DXQUATERNION& );
    D3DXQUATERNION& operator -= ( CONST D3DXQUATERNION& );
    D3DXQUATERNION& operator *= ( CONST D3DXQUATERNION& );
    D3DXQUATERNION& operator *= ( FLOAT );
    D3DXQUATERNION& operator /= ( FLOAT );

    // unary operators
    D3DXQUATERNION  operator + () const;
    D3DXQUATERNION  operator - () const;

    // binary operators
    D3DXQUATERNION operator + ( CONST D3DXQUATERNION& ) const;
    D3DXQUATERNION operator - ( CONST D3DXQUATERNION& ) const;
    D3DXQUATERNION operator * ( CONST D3DXQUATERNION& ) const;
```

```
    D3DXQUATERNION operator * ( FLOAT ) const;
    D3DXQUATERNION operator / ( FLOAT ) const;

    friend D3DXQUATERNION operator * (FLOAT, CONST
D3DXQUATERNION& );

    BOOL operator == ( CONST D3DXQUATERNION& ) const;
    BOOL operator != ( CONST D3DXQUATERNION& ) const;

#endif //__cplusplus
```

Requirements

Header: Declared in D3dx8mesh.h.

D3DXVECTOR2 Extensions

Supplies the following operator overloads and type casts for **D3DXVECTOR2** structures.

```
#ifdef __cplusplus
public:
    D3DXVECTOR2() {};
    D3DXVECTOR2( CONST FLOAT * );
    D3DXVECTOR2( FLOAT x, FLOAT y );

    // casting
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXVECTOR2& operator += ( CONST D3DXVECTOR2& );
    D3DXVECTOR2& operator -= ( CONST D3DXVECTOR2& );
    D3DXVECTOR2& operator *= ( FLOAT );
    D3DXVECTOR2& operator /= ( FLOAT );

    // unary operators
    D3DXVECTOR2 operator + () const;
    D3DXVECTOR2 operator - () const;

    // binary operators
    D3DXVECTOR2 operator + ( CONST D3DXVECTOR2& ) const;
    D3DXVECTOR2 operator - ( CONST D3DXVECTOR2& ) const;
    D3DXVECTOR2 operator * ( FLOAT ) const;
    D3DXVECTOR2 operator / ( FLOAT ) const;
```

```
    friend D3DXVECTOR2 operator * ( FLOAT, CONST D3DXVECTOR2& );

    BOOL operator == ( CONST D3DXVECTOR2& ) const;
    BOOL operator != ( CONST D3DXVECTOR2& ) const;


public:
#endif //__cplusplus
```

Requirements

Header: Declared in D3dx8mesh.h.

D3DXVECTOR3 Extensions

Supplies the following operator overloads and type casts for **D3DXVECTOR3** structures.

```
#ifdef __cplusplus
typedef struct D3DXVECTOR3 : public D3DVECTOR
{
public:
    D3DXVECTOR3() {};
    D3DXVECTOR3( CONST FLOAT * );
    D3DXVECTOR3( CONST D3DVECTOR& );
    D3DXVECTOR3( FLOAT x, FLOAT y, FLOAT z );

    // casting
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXVECTOR3& operator += ( CONST D3DXVECTOR3& );
    D3DXVECTOR3& operator -= ( CONST D3DXVECTOR3& );
    D3DXVECTOR3& operator *= ( FLOAT );
    D3DXVECTOR3& operator /= ( FLOAT );

    // unary operators
    D3DXVECTOR3 operator + () const;
    D3DXVECTOR3 operator - () const;

    // binary operators
    D3DXVECTOR3 operator + ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator - ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator * ( FLOAT ) const;
    D3DXVECTOR3 operator / ( FLOAT ) const;
```

```
    friend D3DXVECTOR3 operator * ( FLOAT, CONST struct
D3DXVECTOR3& );

    BOOL operator == ( CONST D3DXVECTOR3& ) const;
    BOOL operator != ( CONST D3DXVECTOR3& ) const;

} D3DXVECTOR3, *LPD3DXVECTOR3;

#else //!__cplusplus
```

Requirements

Header: Declared in D3dx8mesh.h.

D3DXVECTOR4 Extensions

Supplies the following operator overloads and type casts for **D3DXVECTOR4** structures.

```
#ifdef __cplusplus
public:
    D3DXVECTOR4() {};
    D3DXVECTOR4( CONST FLOAT* );
    D3DXVECTOR4( FLOAT x, FLOAT y, FLOAT z, FLOAT w );

    // casting
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // assignment operators
    D3DXVECTOR4& operator += ( CONST D3DXVECTOR4& );
    D3DXVECTOR4& operator -= ( CONST D3DXVECTOR4& );
    D3DXVECTOR4& operator *= ( FLOAT );
    D3DXVECTOR4& operator /= ( FLOAT );

    // unary operators
    D3DXVECTOR4 operator + () const;
    D3DXVECTOR4 operator - () const;

    // binary operators
    D3DXVECTOR4 operator + ( CONST D3DXVECTOR4& ) const;
    D3DXVECTOR4 operator - ( CONST D3DXVECTOR4& ) const;
    D3DXVECTOR4 operator * ( FLOAT ) const;
    D3DXVECTOR4 operator / ( FLOAT ) const;
```

```
    friend D3DXVECTOR4 operator * ( FLOAT, CONST D3DXVECTOR4& );

    BOOL operator == ( CONST D3DXVECTOR4& ) const;
    BOOL operator != ( CONST D3DXVECTOR4& ) const;

public:
#endif //__cplusplus
```

Requirements

Header: Declared in D3dx8mesh.h.

Return Values

Errors are represented by negative values and cannot be combined. The following lists the values that can be returned by methods included with the Direct3DX utility library. See the individual method descriptions for lists of the values that each can return. These lists are not necessarily comprehensive.

D3DXERR_CANNOTATTRSORT
: Attribute sort (D3DXMESHOPT_ATTRSORT) is not supported as an optimization technique.

D3DXERR_CANNOTMODIFYINDEXBUFFER
: The index buffer cannot be modified.

D3DXERR_INVALIDMESH
: The mesh is invalid.

D3DXERR_SKINNINGNOTSUPPORTED
: Skinning is not supported.

D3DXERR_TOOMANYINFLUENCES
: Too many influences specified.

D3DXERR_INVALIDDATA
: The data is invalid.

X File C/C++ Reference

This section contains reference information for the API elements used to work with Microsoft® DirectX® (.x) files.

- Interfaces
- Functions
- Structures

- Return Values

Interfaces

This section contains reference information for the COM interfaces used to read to and write from Microsoft® DirectX® (.x) files.

- **IDirectXFile**
- **IDirectXFileBinary**
- **IDirectXFileData**
- **IDirectXFileDataReference**
- **IDirectXFileEnumObject**
- **IDirectXFileObject**
- **IDirectXFileSaveObject**

IDirectXFile

Applications use the methods of the **IDirectXFile** interface to create instances of the **IDirectXFileEnumObject** and **IDirectXFileSaveObject** interfaces, and to register templates.

The **IDirectXFile** interface is obtained by calling the **DirectXFileCreate** function.

The methods of the **IDirectXFile** interface can be organized into the following groups.

Creation	**CreateEnumObject**
	CreateSaveObject
Templates	**RegisterTemplates**

The **IDirectXFile** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The globally unique identifier (GUID) for the **IDirectXFile** interface is IID_IDirectXFile.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFile::CreateEnumObject

Creates an enumerator object.

```
HRESULT CreateEnumObject(
  LPVOID pvSource,
  DXFILELOADOPTIONS dwLoadOptions,
  LPDIRECTXFILEENUMOBJECT* ppEnumObj
);
```

Parameters

pvSource
[out] Pointer to data whose contents depend on the value of *dwLoadOptions*

dwLoadOptions
[in] Value that specifies the source of the data. This value can be one of the following flags.

DXFILELOAD_FROMFILE
Indicates data read from a file, so the value in *pvSource* is the name of the file.

DXFILELOAD_FROMRESOURCE
Indicates data read from a resource, so the value in *pvSource* is a **DXFILELOADRESOURCE** structure.

DXFILELOAD_FROMMEMORY
Indicates data read from memory, so the value in *pvSource* is a **DXFILELOADMEMORY** structure.

DXFILELOAD_FROMSTREAM
Indicates data read from a stream. Not currently supported.

DXFILELOAD_FROMURL
Indicates data read from a URL, so the value in *pvSource* is the name of the URL.

ppEnumObj
[out] Address of a pointer to an **IDirectXFileEnumObject** interface, representing the created enumerator object.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADFILEFLOATSIZE

DXFILEERR_BADFILETYPE

DXFILEERR_BADFILEVERSION

DXFILEERR_BADRESOURCE

DXFILEERR_BADVALUE

DXFILEERR_FILENOTFOUND

DXFILEERR_RESOURCENOTFOUND

DXFILEERR_URLNOTFOUND

Remarks

After using this method, use one of the **IDirectXFileEnumObject** methods to retrieve a data object.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

See Also

IDirectXFileEnumObject

IDirectXFile::CreateSaveObject

Creates a save object.

```
HRESULT CreateSaveObject(
  LPCSTR szFileName,
  DXFILEFORMAT dwFileFormat,
  LPDIRECTXFILESAVEOBJECT* ppSaveObj
);
```

Parameters

szFileName
[in] Pointer to the name of the file to use for saving data.

dwFileFormat
[in] Indicates the format to use when saving the Microsoft® DirectX® file. For more information, see Remarks.

DXFILEFORMAT_BINARY
Indicates a binary file.

DXFILEFORMAT_COMPRESSED
Indicates a compressed file.

DXFILEFORMAT_TEXT
Indicates a text file.

ppSaveObj
[out, retval] Address of a pointer to an **IDirectXFileSaveObject** interface, representing the created save object.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADFILE

DXFILEERR_BADVALUE

Remarks

After using this method, use methods of the **IDirectXFileSaveObject** interface to create data objects and to save templates or data.

The default value for the file format is DXFILEFORMAT_BINARY. The file format values can be combined in a logical **OR** to create compressed text or compressed binary files. If a file is specified as both binary (0) and text (1), it will be saved as a text file because the value will be indistinguishable from the text file format value (0 + 1 = 1). If you indicate that the file format should be text and compressed, the file will first be written out as text and then compressed. However, compressed text files are not as efficient as binary text files, so in most cases you will want to indicate binary and compressed. Setting a file to be compressed without specifying a format will result in a binary, compressed file.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

See Also

IDirectXFileSaveObject

IDirectXFile::RegisterTemplates

Registers custom templates.

```
HRESULT RegisterTemplates(
  LPVOID pvData,
  DWORD cbSize
);
```

Parameters

pvData

[in] Pointer to a buffer consisting of a Microsoft® DirectX® file in text or binary format that contains templates.

cbSize

[in] Size of the buffer pointed to by *pvData*, in bytes.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADFILEFLOATSIZE

DXFILEERR_BADFILETYPE

DXFILEERR_BADFILEVERSION

DXFILEERR_BADVALUE

DXFILEERR_PARSEERROR

Remarks

The following code fragment provides an example call to **RegisterTemplates** and example contents for the buffer to which *pvData* points.

```
TIDirectXFile * pDXFile;

char *szTemplates = "xof 0303txt 0032\
   template SimpleData { \
      <2b934580-9e9a-11cf-ab39-0020af71e433> \
      DWORD item1;DWORD item2;DWORD item3;} \
   template ArrayData { \
      <2b934581-9e9a-11cf-ab39-0020af71e433> \
      DWORD cItems; array DWORD aItem[2][cItems]; [...] } \
   template RestrictedData { \
      <2b934582-9e9a-11cf-ab39-0020af71e433> \
      DWORD item; [SimpleData]}";

hr = pDXFile->RegisterTemplates(szTemplates, strlen(szTemplates));
```

All templates must specify a name and a Universally Unique Identifier (UUID).

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileBinary

Applications use the methods of the **IDirectXFileBinary** interface to read and retrieve information about binary data.

The **IDirectXFileBinary** interface inherits the following **IDirectXFileObject** methods, which can be organized into this group.

Information	**GetId**
	GetName

The methods of the **IDirectXFileBinary** interface can be organized into the following groups.

Information	**GetMimeType**
	GetSize
Reading Data	**Read**

The **IDirectXFileBinary** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The globally unique identifier (GUID) for the **IDirectXFileBinary** interface is IID_IDirectXFileBinary.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileBinary::GetMimeType

Retrieves the Multipurpose Internet Mail Extensions (MIME) type for the binary data.

```
HRESULT GetMimeType(
  LPCSTR* pszMimeType
);
```

Parameters

pszMimeType
[out, retval] Address of a pointer to receive the MIME type string.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be DXFILEERR_BADVALUE.

Remarks

When there is no MIME type specified in a Microsoft® DirectX® file for a binary object, the function will set *pszMimeType* to NULL.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileBinary::GetSize

Retrieves the size of the binary data.

```
HRESULT GetSize(
  DWORD* pcbSize
);
```

Parameters

pcbSize
[out, retval] Pointer to the returned size of the binary data, in bytes.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be DXFILEERR_BADVALUE.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileBinary::Read

Reads the binary data.

```
HRESULT Read(
  LPVOID pvData,
  DWORD cbSize,
  LPDWORD pcbRead
);
```

Parameters

pvData
[out] Pointer to the buffer that receives the data that has been read.

cbSize
[in] Size of the buffer pointed to by *pvData*, in bytes.

pcbRead
[out, retval] Pointer to the number of bytes actually read.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOMOREDATA

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileData

Applications use the methods of the **IDirectXFileData** interface to build or to access the immediate hierarchy of the data object. Template restrictions determine the hierarchy. Data types allowed by the template are called optional members. The optional members are not required, but an object might miss important information without them. These optional members are saved as children of the data object. The children can be another data object, a reference to an earlier data object, or a binary object.

The **IDirectXFileData** interface inherits the following **IDirectXFileObject** methods, which can be organized into this group.

Information	**GetId**
	GetName

The methods of the **IDirectXFileData** interface can be organized into the following groups.

Add Data	**AddBinaryObject**
	AddDataObject
	AddDataReference
Retrieve Data	**GetData**
	GetNextObject
	GetType

The **IDirectXFileData** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The globally unique identifier (GUID) for the **IDirectXFileData** interface is IID_IDirectXFileData.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileData::AddBinaryObject

Creates a binary object and adds it as a child object.

```
HRESULT AddBinaryObject(
  LPCSTR szName,
  const GUID* pguid,
  LPCSTR szMimeType,
  LPVOID pvData,
  DWORD cbSize
);
```

Parameters

szName
[in] Pointer to the name of the object. Specify NULL if the object does not need a name.

pguid

[in] Pointer to the globally unique identifier (GUID) representing the object. Specify NULL if the object does not need a GUID.

szMimeType
[in] Pointer to the object's MIME type.

pvData
[in] Pointer to the object's data.

cbSize
[in] Size of the buffer pointed to by *pvData*, in bytes.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

See Also

IDirectXFileBinary::GetMimeType

IDirectXFileData::AddDataObject

Adds a data object as a child object.

```
HRESULT AddDataObject(
  LPDIRECTXFILEDATA pDataObj
);
```

Parameters

pDataObj
[in] Pointer to an **IDirectXFileData** interface, representing the file data object to add as a child object.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

Remarks

Use the **IDirectXFileSaveObject::CreateDataObject** method to create the **IDirectXFileData** object before calling this method.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

See Also

IDirectXFileSaveObject::CreateDataObject

IDirectXFileData::AddDataReference

Creates and adds a data reference object as a child object.

```
HRESULT AddDataReference(
  LPCSTR szRef,
  const GUID* pguidRef
);
```

Parameters

szRef
[in] Pointer to the name of the referenced data object. This parameter can be NULL if *pguidRef* provides a reference to the globally unique identifier (GUID).

pguidRef
[in] Pointer to the GUID representing the data. This parameter can be NULL if *szRef* provides a reference to the name.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

Remarks

For this method to succeed, either the *szRef* or *pguidRef* parameter must be non-NULL.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileData::GetData

Retrieves the data for one of the object's members or the data for all members.

```
HRESULT GetData(
  LPCSTR szMember,
  DWORD* pcbSize,
  void** ppvData
);
```

Parameters

szMember
[in] Pointer to the name of the member for which to retrieve data. Specify NULL to retrieve all required members' data.

pcbSize
[out] Pointer to receive the *ppvData* buffer size, in bytes.

ppvData
[out] Address of a pointer to the buffer to receive the data associated with *szMember*. If *szMember* is NULL, *ppvData* is set to point to a buffer containing all required members' data in a contiguous block of memory.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADARRAYSIZE

DXFILEERR_BADDATAREFERENCE

DXFILEERR_BADVALUE

Remarks

This method retrieves the data for required members of a data object but no data for optional (child) members. Use **IDirectXFileData::GetNextObject** to retrieve child objects.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

See Also

IDirectXFileData::GetNextObject

IDirectXFileData::GetNextObject

Retrieves the next child data object, data reference object, or binary object in the Microsoft® DirectX® file.

```
HRESULT GetNextObject(
  LPDIRECTXFILEOBJECT* ppChildObj
);
```

Parameters

ppChildObj
[out, retval] Address of a pointer to an **IDirectXFileObject** interface, representing the returned child object's file object interface.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOMOREOBJECTS

Remarks

To determine the type of object retrieved, use **QueryInterface** to query the retrieved object for support of **IDirectXFileData**, **IDirectXFileDataReference**, or **IDirectXFileBinary** interfaces. The interface supported indicates the type of object (data, data reference, or binary).

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

See Also

IDirectXFileBinary, **IDirectXFileData**, **IDirectXFileDataReference**

IDirectXFileData::GetType

Retrieves the globally unique identifier (GUID) of the object's template.

```
HRESULT GetType(
  const GUID** ppguid
);
```

Parameters

ppguid
[out, retval] Address of a pointer to receive the GUID of the object's template.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be DXFILEERR_BADVALUE.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileDataReference

Applications use the methods of the **IDirectXFileDataReference** interface to support data reference objects. A data reference object refers to a data object that is defined earlier in the file. This enables you to use the same object multiple times without repeating it in the file.

The **IDirectXFileDataReference** interface inherits the following **IDirectXFileObject** methods, which can be organized into this group.

Information	**GetId**
	GetName

The **IDirectXFileDataReference** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

After you have determined that an object is a data reference object, use the **IDirectXFileDataReference::Resolve** method to retrieve the referenced object defined earlier in the file.

For information on how to identify a data reference object, see the **IDirectXFileData** interface.

The globally unique identifier (GUID) for the **IDirectXFileDataReference** interface is IID_IDirectXFileDataReference.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileDataReference::Resolve

Resolves data references.

```
HRESULT Resolve(
  LPDIRECTXFILEDATA* ppDataObj
);
```

Parameters

ppDataObj
[out, retval] Address of a pointer to an **IDirectXFileData** interface, representing the returned file data object.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOTFOUND

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileEnumObject

Applications use the methods of the **IDirectXFileEnumObject** interface to cycle through the data objects in the file and to retrieve a data object by its globally unique identifier (GUID) or by its name.

The methods of the **IDirectXFileEnumObject** interface can be organized into the following groups.

Object Retrieval	**GetDataObjectById**
	GetDataObjectByName
Enumeration	**GetNextDataObject**

The **IDirectXFileEnumObject** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The GUID for the **IDirectXFileEnumObject** interface is IID_IDirectXFileEnumObject.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileEnumObject::GetDataObjectById

Retrieves the data object that has the specified globally unique identifier (GUID).

```
HRESULT GetDataObjectById(
  REFGUID rguid,
  LPDIRECTXFILEDATA* ppDataObj
);
```

Parameters

rguid
[in] Reference to the requested globally unique identifier (GUID).

ppDataObj
[out] Address of a pointer to an **IDirectXFileData** interface, representing the returned file data object.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOTFOUND

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileEnumObject::GetDataObjectByName

Retrieves the data object that has the specified name.

```
HRESULT GetDataObjectByName(
  LPCSTR szName,
  LPDIRECTXFILEDATA* ppDataObj
);
```

Parameters

szName
[in] Pointer to the requested name.

ppDataObj
[out] Address of a pointer to an **IDirectXFileData** interface, representing the returned file data object.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOTFOUND

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileEnumObject::GetNextDataObject

Retrieves the next top-level object in the Microsoft® DirectX® file.

```
HRESULT GetNextDataObject(
  LPDIRECTXFILEDATA* ppDataObj
);
```

Parameters

ppDataObj
[out] Address of a pointer to an **IDirectXFileData** interface, representing the returned file data object.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOMOREOBJECTS

Remarks

Top-level objects are always data objects. Data reference objects and binary objects can only be children of data objects.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileObject

Applications use the methods of the **IDirectXFileObject** interface to retrieve information about Microsoft® DirectX® file objects.

The methods of the **IDirectXFileObject** interface can be organized into the following group.

Information	**GetId**
	GetName

The **IDirectXFileObject** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The globally unique identifier (GUID) for the **IDirectXFileObject** interface is IID_IDirectXFileObject.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileObject::GetId

Retrieves a pointer to the globally unique identifier (GUID) that identifies a Microsoft® DirectX® file object.

```
HRESULT GetId(
  LPGUID pGuid
);
```

Parameters

pGuid
[out, retval] Pointer to a globally unique identifier (GUID) to receive the object's ID. The function will set the GUID to a NULL GUID if the object does not have an ID.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be DXFILEERR_BADVALUE.

Applies To

This method applies to the following interfaces, which inherit from **IDirectXFileObject**.

- **IDirectXFileBinary**
- **IDirectXFileData**

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileObject::GetName

Retrieves a pointer to a Microsoft® DirectX® file object's name.

```
HRESULT GetName(
  LPSTR pstrNameBuf,
  LPDWORD pdwBufLen
);
```

Parameters

pstrNameBuf
[out] Pointer to the buffer in which the Microsoft® DirectX® file object's name will be copied. Set to NULL if only the buffer length is needed.

pdwBufLen
[in, out] Pointer to a **DWORD** specifying the length of the buffer pointed to by *pstrNameBuf*. The *pdwBufLen* parameter value will be modified to the buffer length needed to hold the object's name even if *pstrNameBuf* is NULL. In either case, the function will return DXFILEERR_BADVALUE if the original value of *pdwBufLen* is not as large as or larger than the length needed to hold the object's name.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

Applies To

This method applies to the following interfaces, which inherit from **IDirectXFileObject**.

- **IDirectXFileBinary**
- **IDirectXFileData**

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileSaveObject

Applications use the methods of the **IDirectXFileSaveObject** interface to create data objects and to save templates and data objects. Note that templates are not required in every file. For example, you could put all templates into a single Microsoft® DirectX® file rather than duplicating them in every DirectX file.

The **IDirectXFileSaveObject** interface is obtained by calling the **IDirectXFile::CreateSaveObject** method.

The methods of the **IDirectXFileSaveObject** interface can be organized into the following group.

Creation	**CreateDataObject**
Saving	**SaveData**
	SaveTemplates

The **IDirectXFileSaveObject** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods.

IUnknown	**AddRef**
	QueryInterface
	Release

The globally unique identifier (GUID) for the **IDirectXFileSaveObject** interface is IID_IDirectXFileSaveObject.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileSaveObject::CreateDataObject

Creates a data object.

```
HRESULT CreateDataObject(
  REFGUID rguidTemplate,
  LPCSTR szName,
  const GUID* pguid,
  DWORD cbSize,
  LPVOID pvData,
  LPDIRECTXFILEDATA* ppDataObj
);
```

Parameters

rguidTemplate
[in] Globally unique identifier (GUID) representing the data object's template.

szName
[in] Pointer to the name of the data object. Specify NULL if the object does not have a name.

pguid
[in] Pointer to a GUID representing the data object. Specify NULL if the object does not have a GUID.

cbSize
[in] Size of the data object, in bytes.

pvData
[in] Pointer to a buffer containing all required member's data.

ppDataObj
[out, retval] Address of a pointer to an **IDirectXFileData** interface, representing the created file data object.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

Remarks

If a data reference object will reference the data object, either the *szName* or *pguid* parameter must be non-NULL.

Save any templates by using the **IDirectXFileSaveObject::SaveTemplates** method before saving the data created by this method. Save the created data by using the **IDirectXFileSaveObject::SaveData** method.

If you need to save optional data, use the **IDirectXFileData::AddDataObject** method after using this method and before using **SaveData**. If the object has child objects, add them before calling **SaveData**.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

See Also

IDirectXFileData::AddDataObject, **IDirectXFileSaveObject::SaveData**, **IDirectXFileSaveObject::SaveTemplates**

IDirectXFileSaveObject::SaveData

Saves a data object and its children to a Microsoft® DirectX® file.

```
HRESULT SaveData(
  LPDIRECTXFILEDATA pDataObj
);
```

Parameters

pDataObj
[in] Pointer to an **IDirectXFileData** interface, representing the file data object to save.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be one of the following values.

DXFILEERR_BADARRAYSIZE

DXFILEERR_BADVALUE

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

IDirectXFileSaveObject::SaveTemplates

Saves templates to a Microsoft® DirectX® file.

```
HRESULT SaveTemplates(
  DWORD cTemplates,
  const GUID** ppguidTemplates
);
```

Parameters

cTemplates
[in] Total number of templates to save.

ppguidTemplates
[in] Address of a pointer to an array of the globally unique identifiers (GUIDs) for all templates to save.

Return Values

If the method succeeds, the return value is DXFILE_OK.

If the method fails, the return value can be DXFILEERR_BADVALUE.

Remarks

The following code fragment provides an example call to **SaveTemplates** and example contents for the array to which *ppguidTemplates* points.

```
IDirectXFileSaveObject * pDXFileSaveObject;

const GUID *aIds[] = {
   &DXFILEOBJ_SimpleData,
   &DXFILEOBJ_ArrayData,
   &DXFILEOBJ_RestrictedData};

hr = pDXFileSaveObject->SaveTemplates(3, aIds);
```

After using this method to save the templates, use the **IDirectXFileSaveObject::CreateDataObject** method to create a data object.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

See Also

IDirectXFileSaveObject::CreateDataObject

Functions

This section contains reference information for the functions that you need to implement when you work with Microsoft® DirectX® files. The following function is implemented.

- **DirectXFileCreate**

DirectXFileCreate

Creates an instance of a DirectXFile object.

```
STDAPI DirectXFileCreate(
  LPDIRECTXFILE* lplpDirectXFile
);
```

Parameters

lplpDirectXFile
Address of a pointer to an **IDirectXFile** interface, representing the created DirectXFile object.

Return Values

If the function succeeds, the return value is DXFILE_OK.

If the function fails, the return value can be one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

Remarks

After using this function, use **IDirectXFile::RegisterTemplates** to register templates, **IDirectXFile::CreateEnumObject** to create an enumerator object, or **IDirectXFile::CreateSaveObject** to create a save object.

Requirements

Header: Declared in Dxfile.h.
Import Library: Use D3dxof.lib.

See Also

IDirectXFile::CreateEnumObject, **IDirectXFile::CreateSaveObject**, **IDirectXFile::RegisterTemplates**

Structures

This section contains information about the structures used with Microsoft® DirectX® files.

- **DXFILELOADMEMORY**
- **DXFILELOADRESOURCE**

DXFILELOADMEMORY

Identifies memory data.

```
typedef struct _DXFILELOADMEMORY {
    LPVOID lpMemory;
    DWORD dSize;
} DXFILELOADMEMORY, *LPDXFILELOADMEMORY;
```

Members

lpMemory
Pointer to a block of memory to be loaded.

dSize
Size of the block of memory to be loaded, in bytes.

Remarks

This structure identifies a resource to be loaded when an application uses the **IDirectXFile::CreateEnumObject** method and specifies DXFILELOAD_FROMMEMORY.

Requirements

Header: Declared in Dxfile.h.

See Also

IDirectXFile::CreateEnumObject

DXFILELOADRESOURCE

Identifies resource data.

```
typedef struct _DXFILELOADRESOURCE {
    HMODULE hModule;
    LPCTSTR lpName;
    LPCTSTR lpType;
} DXFILELOADRESOURCE, *LPDXFILELOADRESOURCE;
```

Members

hModule
Handle of the module containing the resource to be loaded. If this member is NULL, the resource must be attached to the executable file that will use it.

lpName
Pointer to a string specifying the name of the resource to be loaded. For example, if the resource is a mesh, this member should specify the name of the mesh file.

lpType
Pointer to a string specifying the user-defined type identifying the resource.

Remarks

This structure identifies a resource to be loaded when an application uses the **IDirectXFile::CreateEnumObject** method and specifies DXFILELOAD_FROMRESOURCE.

Requirements

Header: Declared in Dxfile.h.

See Also

IDirectXFile::CreateEnumObject

Return Values

The methods used to work with Microsoft® DirectX® (.x) files can return the following values in addition to the standard COM return values.

DXFILE_OK
Command completed successfully.

DXFILEERR_BADALLOC
Memory allocation failed.

DXFILEERR_BADARRAYSIZE
Array size is invalid.

DXFILEERR_BADCACHEFILE
The cache file containing the .x file date is invalid. A cache file contains data retrieved from the network, cached on the hard disk, and retrieved in subsequent requests.

DXFILEERR_BADDATAREFERENCE
Data reference is invalid.

DXFILEERR_BADFILE
File is invalid.

DXFILEERR_BADFILECOMPRESSIONTYPE
File compression type is invalid.

DXFILEERR_BADFILEFLOATSIZE
Floating-point size is invalid.

DXFILEERR_BADFILETYPE
File is not a DirectX (.x) file.

DXFILEERR_BADFILEVERSION
File version is not valid.

DXFILEERR_BADINTRINSICS
Intrinsics are invalid.

DXFILEERR_BADOBJECT
Object is invalid.

DXFILEERR_BADRESOURCE
Resource is invalid.

DXFILEERR_BADTYPE
Object type is invalid.

DXFILEERR_BADVALUE
Parameter is invalid.

DXFILEERR_FILENOTFOUND
File could not be found.

DXFILEERR_INTERNALERROR
Internal error occurred.

DXFILEERR_NOINTERNET
Internet connection not found.

DXFILEERR_NOMOREDATA
No further data is available.

DXFILEERR_NOMOREOBJECTS
All objects have been enumerated.

DXFILEERR_NOMORESTREAMHANDLES
No stream handles are available.

DXFILEERR_NOTDONEYET
Operation has not completed.

DXFILEERR_NOTFOUND
Object could not be found.

DXFILEERR_PARSEERROR
File could not be parsed.

DXFILEERR_RESOURCENOTFOUND
Resource could not be found.

DXFILEERR_NOTEMPLATE
No template available.

DXFILEERR_URLNOTFOUND
URL could not be found.

X File Visual Basic Reference

This section contains reference information for the API elements used to work with Microsoft® DirectX® (.x) files.

- Classes
- Enumerations
- Error Codes

Classes

This section contains reference information for the classes used to read to and write from Microsoft® DirectX® (.x) files.

- **DirectXFile**
- **DirectXFileBinary**
- **DirectXFileData**
- **DirectXFileEnum**
- **DirectXFileObject**
- **DirectXFileReference**
- **DirectXFileSave**

DirectXFile

#Applications use the methods of the **DirectXFile** class to create **DirectXFileEnumObject** and **DirectXFileSaveObject** objects, and to register templates.

The **DirectXFile** class is obtained by calling the **DirectX8.DirectXFileCreate** method.

The methods of the **DirectXFile** class can be organized into the following groups.

Creation	**CreateEnumObject**
	CreateSaveObject
Templates	**RegisterDefaultTemplates**
	RegisterTemplates

The globally unique identifier (GUID) for the **DirectXFile** class is IID_IDirectXFile.

DirectXFile.CreateEnumObject

#Creates an enumerator object.

object.**CreateEnumObject(_**
FileName **As String) As DirectXFileEnum**

Parts

object
Object expression that resolves to a **DirectXFile** object.

FileName
String value set to the file name.

Return Values

Returns a **DirectXFileEnum** object, representing the created enumerator object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADFILEFLOATSIZE

DXFILEERR_BADFILETYPE

IDH_DirectXFile_graphicsxofvb
IDH_DirectXFile.CreateEnumObject_graphicsxofvb

DXFILEERR_BADFILEVERSION

DXFILEERR_BADRESOURCE

DXFILEERR_BADVALUE

DXFILEERR_FILENOTFOUND

DXFILEERR_RESOURCENOTFOUND

DXFILEERR_URLNOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

After using this method, use one of the **DirectXFileEnum** methods to retrieve a data object.

See Also

DirectXFileEnum

DirectXFile.CreateSaveObject

#Creates a save object.

object.**CreateSaveObject(_**
FileName **As String, _**
Flags **As Long) As DirectXFileSave**

Parts

object
Object expression that resolves to a **DirectXFile** object.

FileName
String value, identifying the file to use for saving data.

Flags
A member of the **CONST_DXFILEFORMATFLAGS** enumeration indicating the format to use when saving the DirectX file.

Return Values

Returns a **DirectXFileSave** object, representing the created save object.

IDH_DirectXFile.CreateSaveObject_graphicsxofvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADFILE

DXFILEERR_BADVALUE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

After using this method, use methods of the **DirectXFileSave** class to create data objects and to save templates or data.

The default value for the file format is DXFILEFORMAT_BINARY. The file format values can be combined in a logical **OR** to create compressed text or compressed binary files. If a file is specified as both binary (0) and text (1), it will be saved as a text file because the value will be indistinguishable from the text file format value (0 + 1 = 1). If you indicate that the file format should be text and compressed, the file will first be written out as text and then compressed. However, compressed text files are not as efficient as binary text files, so in most cases you will want to indicate binary and compressed. Setting a file to be compressed without specifying a format will result in a binary, compressed file.

See Also

DirectXFileSave

DirectXFile.RegisterDefaultTemplates

#Registers the default templates.

object.**RegisterDefaultTemplates()**

Parts

object

Object expression that resolves to a **DirectXFile** object.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

IDH_DirectXFile.RegisterDefaultTemplates_graphicsxofvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

DirectXFile.RegisterTemplates

#Registers custom templates.

object.**RegisterTemplates(_**
TemplateData **As Any, _**
Size **As Long)**

Parts

object
Object expression that resolves to a **DirectXFile** object.

TemplateData
Buffer consisting of a Microsoft® DirectX® file in text or binary format that contains templates.

Size
Size of the *TemplateData* buffer, in bytes.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADFILEFLOATSIZE

DXFILEERR_BADFILETYPE

DXFILEERR_BADFILEVERSION

DXFILEERR_BADVALUE

DXFILEERR_PARSEERROR

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

All templates must specify a name and a Universally Unique Identifier (UUID).

DirectXFileBinary

#Applications use the methods of the **DirectXFileBinary** class to read and retrieve information about binary data.

IDH_DirectXFile.RegisterTemplates_graphicsxofvb
IDH_DirectXFileBinary_graphicsxofvb

The **DirectXFileBinary** class implements the following **DirectXFileObject** methods, which can be organized into this group.

Information	**GetId**
	GetName

The methods of the **IDirectXFileBinary** object can be organized into the following groups.

Information	**GetMimeType**
	GetSize
Reading Data	**Read**

The globally unique identifier (GUID) for the **DirectXFileBinary** class is IID_IDirectXFileBinary.

DirectXFileBinary.GetMimeType

#Retrieves the Multipurpose Internet Mail Extensions (MIME) type for the binary data.

object.**GetMimeType() As String**

Parts

object
Object expression that resolves to a **DirectXFileBinary** object.

Return Values

Returns a **String** value set to the file's MIME type.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DXFILEERR_BADVALUE.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

When there is no MIME type specified in a Microsoft® DirectX® file for a binary object, the function will return an empty string.

IDH_DirectXFileBinary.GetMimeType_graphicsxofvb

DirectXFileBinary.GetSize

#Retrieves the size of the binary data.

object.**GetSize() As Long**

Parts

object
Object expression that resolves to a **DirectXFileBinary** object.

Return Values

Returns a **Long** value set to the size of the binary data, in bytes.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DXFILEERR_BADVALUE.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

DirectXFileBinary.Read

#Reads the binary data.

object.**Read(_**
Data **As Any, _**
Size **As Long) As Long**

Parts

object
Object expression that resolves to a **DirectXFileBinary** object.

Data
Buffer that receives the data that has been read.

Size
Long value containing the size of the *Data* buffer, in bytes.

Return Values

Returns a **Long** value set to the number of bytes actually read.

IDH_DirectXFileBinary.GetSize_graphicsxofvb
IDH_DirectXFileBinary.Read_graphicsxofvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOMOREDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

DirectXFileData

#Applications use the methods of the **DirectXFileData** class to build or to access the immediate hierarchy of the data object. Template restrictions determine the hierarchy. Data types allowed by the template are called optional members. The optional members are not required, but an object might miss important information without them. These optional members are saved as children of the data object. The children can be another data object, a reference to an earlier data object, or a binary object.

The **DirectXFileData** class implements the following **DirectXFileObject** methods, which can be organized into this group.

Information	**GetId**
	GetName

The methods of the **DirectXFileData** class can be organized into the following groups.

Add Data	**AddBinaryObject**
	AddDataObject
	AddDataReference
Information	**GetDataSize**
	GetType
Retrieve Data	**GetData**
	GetDataFromOffset
	GetNextObject

The globally unique identifier (GUID) for the **DirectXFileData** class is IID_IDirectXFileData.

IDH_DirectXFileData_graphicsxofvb

DirectXFileData.AddBinaryObject

#Creates a binary object and adds it as a child object.

object.**AddBinaryObject(_**
Name **As String, _**
GuidObject **As String, _**
MimeType **As String, _**
Data **As Any, _**
Size **As Long)**

Parts

object
Object expression that resolves to a **DirectXFileData** object.

Name
String value containing the object's name. Specify an empty string if the object does not need a name.

GuidObject
String value containing the globally unique identifier (GUID) that represents the object. Specify an empty string if the object does not need a GUID.

MimeType
String value containing the object's MIME type.

Data
Buffer containing the object's data.

Size
Long value containing the size of the *Data* buffer, in bytes.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

DirectXFileBinary.GetMimeType

IDH_DirectXFileData.AddBinaryObject_graphicsxofvb

DirectXFileData.AddDataObject

#Adds a data object as a child object.

object.**AddDataObject(_**
Data **As DirectXFileData)**

Parts

object
Object expression that resolves to a **DirectXFileData** object.

Data
DirectXFileData object, representing the file data object to add as a child object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Use the **DirectXFileSave.CreateDataObject** method to create the **DirectXFileData** object before calling this method.

See Also

DirectXFileSave.CreateDataObject

DirectXFileData.AddDataReference

#Creates and adds a data reference object as a child object.

object.**AddDataReference(_**
Name **As String, _**
Guid **As String)**

Parts

object
Object expression that resolves to a **DirectXFileData** object.

IDH_DirectXFileData.AddDataObject_graphicsxofvb
IDH_DirectXFileData.AddDataReference_graphicsxofvb

Name
String value containing the name of the referenced data object. This parameter can be an empty string if *Guid* provides a reference to the globally unique identifier (GUID).

Guid
String value containing the GUID that represents the data. This parameter can be an empty string if *Name* provides a reference to the name.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For this method to succeed, either the *Name* or *Guid* parameter must be a non-empty string.

DirectXFileData.GetData

#Retrieves the data for one of the object's members or the data for all members.

object.**GetData(_**
Name **As String, _**
Data **As Any)**

Parts

object
Object expression that resolves to a **DirectXFileData** object.

Name
String value containing the name of the member for which to retrieve data. Specify an empty string to retrieve all required members' data.

Data
Buffer to receive the data associated with *Name*. If *Name* is an empty string, *Data* is set to a buffer containing all required members' data in a contiguous block of memory.

IDH_DirectXFileData.GetData_graphicsxofvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADARRAYSIZE

DXFILEERR_BADDATAREFERENCE

DXFILEERR_BADVALUE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method retrieves the data for required members of a data object but no data for optional (child) members. Use **DirectXFileData.GetNextObject** to retrieve child objects.

See Also

DirectXFileData.GetNextObject

DirectXFileData.GetDataFromOffset

#Retrieves data starting from a specified offset.

object.**GetDataFromOffset(_**
Name **As String, _**
Offset **As Long, _**
ByteCount **As Long, _**
Data **As Any)**

Parts

object
Object expression that resolves to a **DirectXFileData** object.

Name
String value containing the name of the member for which to retrieve data. Specify an empty string to retrieve all required members' data.

Offset
Long value containing the offset into the data buffer.

ByteCount
The size of the data to retrieve, in bytes.

Data

IDH_DirectXFileData.GetDataFromOffset_graphicsxofvb

Buffer to receive the data associated with *Name*. If *Name* is an empty string, *Data* is set to a buffer containing all required members' data in a contiguous block of memory.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

DirectXFileData.GetDataSize

DirectXFileData.GetDataSize

#Retrieves the data size of the specified member.

object.**GetDataSize(_**
Name **As String) As Long**

Parts

object
Object expression that resolves to a **DirectXFileData** object.

Name
String value containing the name of the member for which to retrieve the data size.

Return Values

Returns a **Long** value that is set to the data size, in bytes.

Error Codes

If the method fails, an error is raised and **Err.Number** is set.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

DirectXFileData.GetNextObject

#Retrieves the next child data object, data reference object, or binary object in the Microsoft® DirectX® file.

IDH_DirectXFileData.GetDataSize_graphicsxofvb
IDH_DirectXFileData.GetNextObject_graphicsxofvb

object.**GetNextObject() As DirectXFileObject**

Parts

object
Object expression that resolves to a **DirectXFileData** object.

Return Values

Returns a **DirectXFileObject** object, representing the returned child object's file object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOMOREOBJECTS

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

DirectXFileData.GetType

#Retrieves the globally unique identifier (GUID) of the object's template.

object.**GetType() As String**

Parts

object
Object expression that resolves to a **DirectXFileData** object.

Return Values

Returns a **String** value containing the GUID of the object's template.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DXFILEERR_BADVALUE.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_DirectXFileData.GetType_graphicsxofvb

DirectXFileEnum

#Applications use the methods of the **DirectXFileEnum** class to cycle through the data objects in the file and to retrieve a data object by its globally unique identifier (GUID) or by its name.

The methods of the **DirectXFileEnum** class can be organized into the following groups.

Object Retrieval	**GetDataObjectById**
	GetDataObjectByName
Enumeration	**GetNextDataObject**

The globally unique identifier (GUID) for the **DirectXFileEnum** class is IID_IDirectXFileEnumObject.

DirectXFileEnum.GetDataObjectById

#Retrieves a data object specified by its globally unique identifier (GUID).

object.**GetDataObjectById(_**
ID **As String, _**
RetVal **As DirectXFileData)**

Parts

object
Object expression that resolves to a **DirectXFileEnum** object.

ID
String value set to the object's GUID.

RetVal
DirectXFileData object, representing the returned file data object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_DirectXFileEnum_graphicsxofvb
IDH_DirectXFileEnum.GetDataObjectById_graphicsxofvb

DirectXFileEnum.GetDataObjectByName

#Retrieves the data object that has the specified name.

object.**GetDataObjectByName(_**
ID **As String, _**
RetVal **As DirectXFileData)**

Parts

object
Object expression that resolves to a **DirectXFileEnum** object.

ID
String value containing the object's name.

RetVal
DirectXFileData object, representing the returned file data object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

DirectXFileEnum.GetNextDataObject

#Retrieves the next top-level object in the Microsoft® DirectX® file.

object.**GetNextDataObject() As DirectXFileData**

Parts

object
Object expression that resolves to a **DirectXFileEnum** object.

Return Values

Returns a **DirectXFileData** object, representing the returned file data object.

IDH_DirectXFileEnum.GetDataObjectByName_graphicsxofvb
IDH_DirectXFileEnum.GetNextDataObject_graphicsxofvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOMOREOBJECTS

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Top-level objects are always data objects. Data reference objects and binary objects can only be children of data objects.

DirectXFileObject

#Applications use the methods of the **DirectXFileObject** class to retrieve information about Microsoft® DirectX® file objects.

The methods of the **DirectXFileObject** class can be organized into the following group.

Information	**GetId**
	GetName

The globally unique identifier (GUID) for the **DirectXFileObject** class is IID_IDirectXFileObject.

DirectXFileObject.GetId

#Retrieves the globally unique identifier (GUID) that identifies a Microsoft® DirectX® file object.

object.**GetId() As String**

Parts

object

Object expression that resolves to a **DirectXFileObject** object.

Return Values

Returns a **String** containing the object's GUID.

IDH_DirectXFileObject_graphicsxofvb
IDH_DirectXFileObject.GetId_graphicsxofvb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DXFILEERR_BADVALUE.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **DirectXFileObject**.

- **DirectXFileBinary**
- **DirectXFileData**

Remarks

Returns a **String** value containing the file's GUID.

DirectXFileObject.GetName

#Retrieves a Microsoft® DirectX® file object's name.

object.**GetName() As String**

Parts

object
Object expression that resolves to a **DirectXFileObject** object.

Return Values

Returns a **String** value containing the object's name.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_DirectXFileObject.GetName_graphicsxofvb

Applies To

This method applies to the following classes, which implement methods from **DirectXFileObject**.

- **DirectXFileBinary**
- **DirectXFileData**

DirectXFileReference

#Applications use the methods of the **DirectXFileReference** class to support data reference objects. A data reference object refers to a data object that is defined earlier in the file. This enables you to use the same object multiple times without repeating it in the file.

The **DirectXFileReference** class implements the following **DirectXFileObject** methods, which can be organized into this group.

Information	**GetId**
	GetName

After you have determined that an object is a data reference object, use the **DirectXFileReference.Resolve** method to retrieve the referenced object defined earlier in the file.

For information on how to identify a data reference object, see the **DirectXFileData** class.

The globally unique identifier (GUID) for the **DirectXFileReference** class is IID_IDirectXFileDataReference.

DirectXFileReference.Resolve

#Resolves data references.

object.**Resolve() As DirectXFileData**

Parts

object

Object expression that resolves to a **DirectXFileReference** object.

Return Values

DirectXFileData object, representing the returned file data object.

IDH_DirectXFileReference_graphicsxofvb
IDH_DirectXFileReference.Resolve_graphicsxofvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADVALUE

DXFILEERR_NOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

DirectXFileSave

#Applications use the methods of the **DirectXFileSave** class to create data objects and to save templates and data objects. Note that templates are not required in every file. For example, you could put all templates into a single Microsoft® DirectX® file rather than duplicating them in every DirectX file.

The **DirectXFileSave** class is obtained by calling the **DirectXFile.CreateSaveObject** method.

The methods of the **DirectXFileSave** class can be organized into the following group.

Creation	**CreateDataObject**
Saving	**SaveData**
	SaveTemplates

The globally unique identifier (GUID) for the **DirectXFileSave** class is IID_IDirectXFileSaveObject.

DirectXFileSave.CreateDataObject

#Creates a data object.

object.**CreateDataObject(_**
TemplateGuid **As String, _**
Name **As String, _**
DataTypeGuid **As String, _**
ByteCount **As Long, _**
Data **As Any) As DirectXFileData**

Parts

object
Object expression that resolves to a **DirectXFileSave** object.

IDH_DirectXFileSave_graphicsxofvb
IDH_DirectXFileSave.CreateDataObject_graphicsxofvb

TemplateGuid
String value containing the data object's template globally unique identifier (GUID).

Name
String value containing the object's name. Specify an empty string if the object does not have a name.

DataTypeGuid
String value containing the data object's GUID. Specify an empty string if the object does not have a GUID.

ByteCount
Long value containing the size of the data object, in bytes.

Data
Buffer containing all required member's data.

Return Values

DirectXFileData object, representing the created file data object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADALLOC

DXFILEERR_BADVALUE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If a data reference object will reference the data object, either the *Name* or *DataTypeGuid* parameter must be a non-empty string.

Save any templates by using the **DirectXFileSave.SaveTemplates** method before saving the data created by this method. Save the created data by using the **DirectXFileSave.SaveData** method.

If you need to save optional data, use the **DirectXFileData.AddDataObject** method after using this method and before using **SaveData**. If the object has child objects, add them before calling **SaveData**.

See Also

DirectXFileData.AddDataObject, **DirectXFileSave.SaveData**, **DirectXFileSave.SaveTemplates**

DirectXFileSave.SaveData

#Saves a data object and its children to a Microsoft® DirectX® file.

object.**SaveData(_**
DataObj **As DirectXFileData)**

Parts

object
Object expression that resolves to a **DirectXFileSave** object.

DataObj
DirectXFileData object, representing the file data object to save.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

DXFILEERR_BADARRAYSIZE

DXFILEERR_BADVALUE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

DirectXFileSave.SaveTemplates

#Saves templates to a Microsoft® DirectX® file.

object.**SaveTemplates(_**
Count **As Long, _**
TemplateGuids() **As String)**

Parts

object
Object expression that resolves to a **DirectXFileSave** object.

Count
Total number of templates to save.

TemplateGuids
Array of **String** values containing the GUIDs for all templates to save.

IDH_DirectXFileSave.SaveData_graphicsxofvb
IDH_DirectXFileSave.SaveTemplates_graphicsxofvb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DXFILEERR_BADVALUE.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

DirectXFileSave.CreateDataObject

Enumerations

This section contains information about the enumerations used with DirectX® files.

- **CONST_DXFILEERR**
- **CONST_DXFILEFORMATFLAGS**

CONST_DXFILEERR

#Defines file error codes raised by the system. For descriptions of these error codes, see Error Codes.

CONST_DXFILEFORMATFLAGS

#Defines constants that describe the fill mode.

```
Enum CONST_DXFILEFORMATFLAGS
   DXFILEFORMAT_BINARY     = 0
   DXFILEFORMAT_TEXT       = 1
   DXFILEFORMAT_COMPRESSED = 2
End Enum
```

Constants

DXFILEFORMAT_BINARY
Indicates a binary file.

DXFILEFORMAT_COMPRESSED
Indicates a compressed file.

DXFILEFORMAT_TEXT
Indicates a text file.

IDH_CONST_DXFILEERR_graphicsxofvb
IDH_CONST_DXFILEFORMATFLAGS_graphicsxofvb

See Also

D3DX8.SaveMeshToX, DirectXFile.CreateSaveObject

Error Codes

The methods used to work with Microsoft® DirectX (.x) files can return the following error codes in addition to standard error codes.

DXFILE_OK
Command completed successfully.

DXFILEERR_BADALLOC
Memory allocation failed.

DXFILEERR_BADARRAYSIZE
Array size is invalid.

DXFILEERR_BADCACHEFILE
The cache file containing the .x file date is invalid. A cache file contains data retrieved from the network, cached on the hard disk, and retrieved in subsequent requests.

DXFILEERR_BADDATAREFERENCE
Data reference is invalid.

DXFILEERR_BADFILE
File is invalid.

DXFILEERR_BADFILECOMPRESSIONTYPE
File compression type is invalid.

DXFILEERR_BADFILEFLOATSIZE
Floating-point size is invalid.

DXFILEERR_BADFILETYPE
File is not a DirectX (.x) file.

DXFILEERR_BADFILEVERSION
File version is not valid.

DXFILEERR_BADINTRINSICS
Intrinsics are invalid.

DXFILEERR_BADOBJECT
Object is invalid.

DXFILEERR_BADRESOURCE
Resource is invalid.

DXFILEERR_BADTYPE
Object type is invalid.

DXFILEERR_BADVALUE
Parameter is invalid.

DXFILEERR_FILENOTFOUND
File could not be found.

DXFILEERR_INTERNALERROR
Internal error occurred.

DXFILEERR_NOINTERNET
Internet connection not found.

DXFILEERR_NOMOREDATA
No further data is available.

DXFILEERR_NOMOREOBJECTS
All objects have been enumerated.

DXFILEERR_NOMORESTREAMHANDLES
No stream handles are available.

DXFILEERR_NOTDONEYET
Operation has not completed.

DXFILEERR_NOTFOUND
Object could not be found.

DXFILEERR_PARSEERROR
File could not be parsed.

DXFILEERR_RESOURCENOTFOUND
Resource could not be found.

DXFILEERR_NOTEMPLATE
No template available.

DXFILEERR_URLNOTFOUND
URL could not be found.

X File Format Reference

This section contains reference information for the Microsoft® DirectX® (.x) file format.

- X File Templates
- Binary Format

X File Templates

This section lists and details the following .x file templates.

- **Header**
- **Vector**
- **Coords2d**
- **Quaternion**
- **Matrix4x4**

- **ColorRGBA**
- **ColorRGB**
- **IndexedColor**
- **Boolean**
- **Boolean2d**
- **Material**
- **TextureFilename**
- **MeshFace**
- **MeshFaceWraps**
- **MeshTextureCoords**
- **MeshNormals**
- **MeshVertexColors**
- **MeshMaterialList**
- **Mesh**
- **FrameTransformMatrix**
- **Frame**
- **FloatKeys**
- **TimedFloatKeys**
- **AnimationKey**
- **AnimationOptions**
- **Animation**
- **AnimationSet**
- **Patch**
- **PatchMesh**
- **VertexDuplicationIndices**
- **XSkinMeshHeader**
- **SkinWeights**

Header

Defines the application-specific header.

UUID

<3D82AB43-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
major	**WORD**		None

minor	**WORD**
flags	**DWORD**

Vector

Defines a vector.

UUID

<3D82AB5E-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
x	**FLOAT**		None
y	**FLOAT**		
z	**FLOAT**		

Coords2d

A two dimensional vector used to define a mesh's texture coordinates.

UUID

<F6F23F44-7686-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
u	**FLOAT**		None
v	**FLOAT**		

Quaternion

Currently unused.

UUID

<10DD46A3-775B-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
s	**FLOAT**		None
v	**Vector**		

Matrix4x4

Defines a 4×4 matrix. This is used as a frame transformation matrix.

UUID

<F6F23F45-7686-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
matrix	array **FLOAT**	16	None

ColorRGBA

Defines a color object with an alpha component. This is used for the face color in the material template definition.

UUID

<35FF44E0-6C7C-11cf-8F52-0040333594A3>

Member name	Type	Optional array Size	Optional data objects
red	**FLOAT**		None
green	**FLOAT**		
blue	**FLOAT**		
alpha	**FLOAT**		

ColorRGB

Defines the basic RGB color object.

UUID

<D3E16E81-7835-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
red	**FLOAT**		None
green	**FLOAT**		
blue	**FLOAT**		

IndexedColor

Consists of an index parameter and an RGBA color and is used for defining mesh vertex colors. The index defines the vertex to which the color is applied.

UUID

<1630B820-7842-11cf-8F52-0040333594A3>

Member name	Type	Optional array size
index	**DWORD**	
indexColor	**ColorRGBA**	

Boolean

Defines a simple Boolean type. This template should be set to 0 or 1.

UUID

<4885AE61-78E8-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
truefalse	**DWORD**		None

Boolean2d

Defines a set of two Boolean values used in the **MeshFaceWraps** template to define the texture topology of an individual face.

UUID

<4885AE63-78E8-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
u	**Boolean**		None
v	**Boolean**		

Material

Defines a basic material color that can be applied to either a complete mesh or a mesh's individual faces. The power is the specular exponent of the material. Note that the ambient color requires an alpha component.

TextureFilename is an optional data object. If this object is not present, the face is untextured.

UUID

<3D82AB4D-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
faceColor	**ColorRGBA**		Any
power	**FLOAT**		
specularColor	**ColorRGB**		
emissiveColor	**ColorRGB**		

TextureFilename

Enables you to specify the file name of a texture to apply to a mesh or a face. This template should appear within a material object.

UUID

<A42790E1-7810-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
filename	**STRING**		None

MeshFace

Used by the **Mesh** template to define a mesh's faces. Each element of the **nFaceVertexIndices** array references a mesh vertex used to build the face.

UUID

<3D82AB5F-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
nFaceVertexIndices	**DWORD**		None
faceVertexIndices	array **DWORD**	**nFaceVertexIndicies**	

MeshFaceWraps

Used to define the texture topology of each face in a wrap. The value of the **nFaceWrapValues** member should be equal to the number of faces in a mesh.

UUID

<4885AE62-78E8-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nFaceWrapValues	**DWORD**		None
faceWrapValues	**Boolean2d**		

MeshTextureCoords

Defines a mesh's texture coordinates.

UUID

<F6F23F40-7686-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nTextureCoords	**DWORD**		None
textureCoords	array **Coords2d**	**nTextureCoords**	

MeshNormals

Defines normals for a mesh. The first array of vectors is the normal vectors themselves, and the second array is an array of indexes specifying which normals should be applied to a given face. The value of the **nFaceNormals** member should be equal to the number of faces in a mesh.

UUID

<F6F23F43-7686-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nNormals	**DWORD**		None
normals	array **Vector**	**nNormals**	
nFaceNormals	**DWORD**		
faceNormals	array **MeshFace**	**nFaceNormals**	

MeshVertexColors

Specifies vertex colors for a mesh, instead of applying a material per face or per mesh.

UUID

<1630B821-7842-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nVertexColors	**DWORD**		None
vertexColors	array **IndexedColor**	**nVertexColors**	

MeshMaterialList

Used in a mesh object to specify which material applies to which faces. The **nMaterials** member specifies how many materials are present, and materials specify which material to apply.

UUID

<F6F23F42-7686-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nMaterials	**DWORD**		**Material**
nFaceIndexes	**DWORD**		
FaceIndexes	array **DWORD**	**nFaceIndexes**	

Mesh

Defines a simple mesh. The first array is a list of vertices, and the second array defines the faces of the mesh by indexing into the vertex array.

UUID

<3D82AB44-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
nVertices	**DWORD**		Any
vertices	array **Vector**	**nVertices**	
nFaces	**DWORD**		

faces	array **MeshFace**	**nFaces**

Optional Data Elements

The following optional data elements can be used.

Data element	Description
MeshFaceWraps	If this is not present, wrapping for both u and v defaults to false.
MeshTextureCoords	If this is not present, there are no texture coordinates.
MeshNormals	If this is not present, normals are generated using the **GenerateNormals** meth of the API.
MeshVertexColors	If this is not present, the colors default to white.
MeshMaterialList	If this is not present, the material defaults to white.

FrameTransformMatrix

Defines a local transform for a frame (and all its child objects).

UUID

<F6F23F41-7686-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
frameMatrix	**Matrix4x4**		None

Frame

Defines a frame. Currently, the frame can contain objects of the type **Mesh** and a **FrameTransformMatrix**.

UUID

<3D82AB46-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
None			Any

Optional Data Elements

The following optional elements can be used.

Data element	Description

FrameTransformMatrix	If this element is not present, no local transform is applied to the frame.
Mesh	Any number of mesh objects that become children of the frame. These obje can be specified inline or by reference.

FloatKeys

Defines an array of floating-point numbers (floats) and the number of floats in that array. This is used for defining sets of animation keys.

UUID

<10DD46A9-775B-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
nValues	**DWORD**		None
values	array **FLOAT**	**nValues**	

TimedFloatKeys

Defines a set of floats and a positive time used in animations.

UUID

<F406B180-7B3B-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
time	**DWORD**		None
tfkeys	**FloatKeys**		

AnimationKey

Defines a set of animation keys. The **keyType** member specifies whether the keys are rotation, scale position, or matrix keys (using the integers 0, 1, 2, or 3 respectively). A matrix key is useful for sets of animation data that need to be represented as transformation matrices.

UUID

<10DD46A8-775B-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects

keyType	**DWORD**		None
nKeys	**DWORD**		
keys	array **TimedFloatKeys**	**nKeys**	

AnimationOptions

Enables you to set the animation options. The **openclosed** member can be either 0 for a closed or 1 for an open animation. The **positionquality** member is used to set the position quality for any position keys specified and can either be 0 for spline positions or 1 for linear positions. By default, an animation is closed.

UUID

<E2BF56C0-840F-11cf-8F52-0040333594A3>

Member name	Type	Optional array size	Optional data objects
openclosed	**DWORD**		None
positionquality	**DWORD**		

Animation

Contains animations referencing a previous frame. It should contain one reference to a frame and at least one set of **AnimationKey** templates. It also can contain an **AnimationOptions** data object.

UUID

<3D82AB4F-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
None			Any

Optional Data Elements

The following optional data elements are used.

Data element	Description
AnimationKey	All animations require AnimationKeys.
AnimationOptions	If this element is not present, an animation is closed.

AnimationSet

Contains one or more **Animation** objects. Each animation within an animation set has the same time at any given point. Increasing the animation set's time increases the time for all the animations it contains.

UUID

<3D82AB50-62DA-11cf-AB39-0020AF71E433>

Member name	Type	Optional array size	Optional data objects
None			**Animation**

Patch

Defines a bézier control patch. The array defines the control points for the patch.

UUID

<A3EB5D44-FC22-429D-9AFB-3221CB9719A6>

Member name	Type	Optional array size	Optional data objects
nControlIndices	**DWORD**		None
controlIndices	array **DWORD**	**nControlIndices**	

The type of patch is defined by the number of control points, as shown in the following table.

Number of control points	Type
10	Cubic Bézier Triangular Patch
15	Quartic Bézier Triangular Patch
16	Cubic Bézier Quad (Rectangular) Patch

The order of the control points are given in a spiral pattern, as shown in the following diagrams for triangular and rectangular patches.

Triangular patches used the following pattern.

Rectangular patches use the following pattern.

PatchMesh

Defines a mesh defined by bézier patches. The first array is a list of vertices, and the second array defines the patches for the mesh by indexing into the vertex array.

UUID

<D02C95CC-EDBA-4305-9B5D-1820D7704BBF>

Member name	Type	Optional array size	Optional data objects
nVertices	**DWORD**		None
vertices	array **Vector**	**nVertices**	
nPatches	**DWORD**		
patches	array **Patch**	**nPatches**	

The patches use the vertices in the array of vertices as the control points for each patch.

VertexDuplicationIndices

This template is instantiated on a per-mesh basis, holding information about which vertices in the mesh are duplicates of each other. Duplicates result when a vertex sits on a smoothing group or material boundary. The purpose of this template is to allow the loader to determine which vertices exhibiting different peripheral parameters are actually the same vertexes in the model. Certain applications (mesh simplification for example) can make use of this information.

UUID

<B8D65549-D7C9-4995-89CF-53A9A8B031E3>

Member name	Type	Optional array size	Optional data objects
nIndices	**DWORD**		None
nOriginalVertices	**DWORD**		
indices	array **DWORD**	**nIndices**	

The number of vertices in the mesh before any duplication occurred is **nOriginalVertices**. The value **indices**[*n*] holds the vertex index that vertex[*n*] in the vertex array for the mesh would have had if no duplication had occurred. So any indices in this array that are the same indicate duplicate vertices.

XSkinMeshHeader

This template is instantiated on a per-mesh basis only in meshes that contain exported skinning information. The purpose of this template is to provide information about the nature of the skinning information that was exported.

UUID

<3CF169CE-FF7C-44ab-93C0-F78F62D172E2>

Member name	Type	Optional array size	Optional data obj
nMaxSkinWeightsPerVertex	**WORD**		None
nMaxSkinWeightsPerFace	**WORD**		
nBones	**WORD**		

The maximum number of transforms that affect a vertex in the mesh is **nMaxSkinWeightsPerVertex**. The maximum number of unique transforms that affect the three vertices of any face is **nMaxSkinWeightsPerFace**. The number of bones that affect vertices in this mesh is **nBones**.

SkinWeights

This template is instantiated on a per-mesh basis. Within a mesh, a sequence of n instances of this template will appear, where *n* is the number of bones (X file frames) that influence the vertices in the mesh. Each instance of the template basically defines the influence of a particular bone on the mesh. There is a list of vertex indices, and a corresponding list of weights.

UUID

<6F0D123B-BAD2-4167-A0D0-80224F25FABB>

Member name	Type	Optional array size	Optional data objects
transformNodeName	**STRING**		None
nWeights	**DWORD**		
vertexIndices	array **DWORD**	**nWeights**	
weights	array float	**nWeights**	
matrixOffset	**Matrix4x4**		

The name of the bone whose influence is being defined is **transformNodeName**, and **nWeights** is the number of vertices affected by this bone. The vertices influenced by this bone are contained in **vertexIndices**, and the weights for each of the vertices influenced by this bone are contained in **weights**.

The matrix **matrixOffset** transforms the mesh vertices to the space of the bone. When concatenated to the bone's transform, this provides the world space coordinates of the mesh as affected by the bone.

Binary Format

This section details the binary version of the Microsoft® DirectX® (.x) file format as introduced with the release of DirectX 3.0. The information presented in this section should be read in conjunction with the X File Architecture section.

The binary format is a tokenized representation of the text format. Tokens can be stand-alone or accompanied by primitive data records. Stand-alone tokens give grammatical structure, and record-bearing tokens supply the necessary data.

Note that all data is stored in little-endian format. A valid binary data stream consists of a header followed by templates and/or data objects.

This section discusses the following components of the binary file format and provides an example template and binary data object.

- Header
- Tokens
- Token Records
- Templates
- Data
- Examples

Header

The following definitions should be used when directly reading and writing the binary header. Note that compressed data streams are not currently supported and are therefore not detailed here.

```
#define XOFFILE_FORMAT_MAGIC \
  ((long)'x' + ((long)'o' << 8) + ((long)'f' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_VERSION \
  ((long)'0' + ((long)'3' << 8) + ((long)'0' << 16) + ((long)'2' << 24))

#define XOFFILE_FORMAT_BINARY \
  ((long)'b' + ((long)'i' << 8) + ((long)'n' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_TEXT   \
  ((long)'t' + ((long)'x' << 8) + ((long)'t' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_COMPRESSED \
```

```
  ((long)'c' + ((long)'m' << 8) + ((long)'p' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_FLOAT_BITS_32 \
  ((long)'0' + ((long)'0' << 8) + ((long)'3' << 16) + ((long)'2' << 24))

#define XOFFILE_FORMAT_FLOAT_BITS_64 \
  ((long)'0' + ((long)'0' << 8) + ((long)'6' << 16) + ((long)'4' << 24))
```

Tokens

Tokens are written as little-endian **WORD**s. The following list of token values is divided into record-bearing and stand-alone tokens.

The record-bearing tokens are defined as follows.

```
#define TOKEN_NAME 1
#define TOKEN_STRING 2
#define TOKEN_INTEGER 3
#define TOKEN_GUID 5
#define TOKEN_INTEGER_LIST 6
#define TOKEN_FLOAT_LIST 7
```

The stand-alone tokens are defined as follows.

```
#define TOKEN_OBRACE 10
#define TOKEN_CBRACE 11
#define TOKEN_OPAREN 12
#define TOKEN_CPAREN 13
#define TOKEN_OBRACKET 14
#define TOKEN_CBRACKET 15
#define TOKEN_OANGLE 16
#define TOKEN_CANGLE 17
#define TOKEN_DOT 18
#define TOKEN_COMMA 19
#define TOKEN_SEMICOLON 20
#define TOKEN_TEMPLATE 31
#define TOKEN_WORD 40
#define TOKEN_DWORD 41
#define TOKEN_FLOAT 42
#define TOKEN_DOUBLE 43
#define TOKEN_CHAR 44
#define TOKEN_UCHAR 45
#define TOKEN_SWORD 46
#define TOKEN_SDWORD 47
#define TOKEN_VOID 48
#define TOKEN_LPSTR 49
#define TOKEN_UNICODE 50
```

```
#define TOKEN_CSTRING 51
#define TOKEN_ARRAY 52
```

Token Records

This section describes the format of the records for each of the record-bearing tokens. Information is divided into the following sections.

- **TOKEN_NAME**
- **TOKEN_STRING**
- **TOKEN_INTEGER**
- **TOKEN_GUID**
- **TOKEN_INTEGER_LIST**
- **TOKEN_FLOAT_LIST**

TOKEN_NAME

A variable-length record. The token is followed by a *count* value that specifies the number of bytes that follow in the *name* field. An ASCII name of length *count* completes the record.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_NAME**
count	**DWORD**	4	Length of name field, in bytes
name	**BYTE** array	**count**	ASCII name

TOKEN_STRING

A variable-length record. The token is followed by a count value that specifies the number of bytes that follow in the string field. An ASCII string of length count continues the record, which is completed by a terminating token. The choice of terminator is determined by syntax issues discussed elsewhere.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_STRING**
count	**DWORD**	4	Length of string field in bytes
string	**BYTE** array	**count**	ASCII string
terminator	**DWORD**	4	**TOKEN_SEMICOLON** or **TOKEN_COMMA**

TOKEN_INTEGER

A fixed length record. The token is followed by the integer value required.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_INTEGER**
value	**DWORD**	4	Single integer

TOKEN_GUID

A fixed-length record. The token is followed by the four data fields as defined by the OSF DCE standard.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_GUID**
data1	**DWORD**	4	UUID data field 1
data2	**WORD**	2	UUID data field 2
data3	**WORD**	2	UUID data field 3
data4	**BYTE** array	8	UUID data field 4

TOKEN_INTEGER_LIST

A variable-length record. The token is followed by a count value that specifies the number of integers that follow in the list field. For efficiency, consecutive integer lists should be compounded into a single list.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_INTEGER_LIST**
count	**DWORD**	4	Number of integers in list field
list	**DWORD**	4 × **count**	Integer list

TOKEN_FLOAT_LIST

A variable-length record. The token is followed by a count value that specifies the number of floats or doubles that follow in the list field. The size of the floating point value (float or double) is determined by the value of float size specified in the file header. For efficiency, consecutive **TOKEN_FLOAT_LIST**s should be compounded into a single list.

Field	Type	Size (bytes)	Contents
token	**WORD**	2	**TOKEN_FLOAT_LIST**

count	**DWORD**	4	Number of floats or doubles in list field
list	float/double array	4 or 8 × **count**	Float or double list

Templates

A template has the following syntax definition.

```
template            : TOKEN_TEMPLATE name TOKEN_OBRACE
                      class_id
                      template_parts
                      TOKEN_CBRACE

template_parts      : template_members_part TOKEN_OBRACKET
                      template_option_info
                      TOKEN_CBRACKET
                    | template_members_list

template_members_part : /* Empty */
                    | template_members_list

template_option_info  : ellipsis
                    | template_option_list

template_members_list :   template_members
                    | template_members_list template_members

template_members    : primitive
                    | array
                    | template_reference

primitive           : primitive_type optional_name TOKEN_SEMICOLON

array               : TOKEN_ARRAY array_data_type name dimension_list
                      TOKEN_SEMICOLON

template_reference  : name optional_name YT_SEMICOLON

primitive_type      : TOKEN_WORD
                    | TOKEN_DWORD
                    | TOKEN_FLOAT
                    | TOKEN_DOUBLE
                    | TOKEN_CHAR
                    | TOKEN_UCHAR
                    | TOKEN_SWORD
                    | TOKEN_SDWORD
```

```
                | TOKEN_LPSTR
                | TOKEN_UNICODE
                | TOKEN_CSTRING

array_data_type     : primitive_type
                | name

dimension_list      : dimension
                | dimension_list dimension

dimension           : TOKEN_OBRACKET dimension_size TOKEN_CBRACKET

dimension_size      : TOKEN_INTEGER
                | name

template_option_list  : template_option_part
                | template_option_list template_option_part

template_option_part  : name optional_class_id

name                : TOKEN_NAME

optional_name       : /* Empty */
                | name

class_id            : TOKEN_GUID

optional_class_id   : /* Empty */
                | class_id

ellipsis            : TOKEN_DOT TOKEN_DOT TOKEN_DOT
```

Data

A data object has the following syntax definition.

```
object              : identifier optional_name TOKEN_OBRACE
                      optional_class_id
                      data_parts_list
                      TOKEN_CBRACE
data_parts_list     : data_part
                | data_parts_list data_part

data_part           : data_reference
                | object
```

```
                | number_list
                | float_list
                | string_list

number_list         : TOKEN_INTEGER_LIST

float_list          : TOKEN_FLOAT_LIST

string_list         : string_list_1 list_separator

string_list_1       : string
                | string_list_1 list_separator string

list_separator      : comma
                | semicolon

string              : TOKEN_STRING

identifier          : name
                | primitive_type

data_reference      : TOKEN_OBRACE name optional_class_id TOKEN_CBRACE
```

Note that in number_list and float_list data in binary files, TOKEN_COMMA and TOKEN_SEMICOLON are not used. The comma and semicolon are used in string_list data. Also note that you can only use data_reference for optional data members.

Examples

Two example binary template definitions and an example of a binary data object follow. Note that data is stored in little-endian format, which is not shown in these examples.

The closed template *RGB* is identified by the UUID {55b6d780-37ec-11d0-ab39-0020af71e433} and has three members *r*, *g*, and *b* each of type *float*.

```
TOKEN_TEMPLATE, TOKEN_NAME, 3, 'R', 'G', 'B', TOKEN_OBRACE,
TOKEN_GUID, 55b6d780, 37ec, 11d0, ab, 39, 00, 20, af, 71, e4, 33,
TOKEN_FLOAT, TOKEN_NAME, 1, 'r', TOKEN_SEMICOLON,
TOKEN_FLOAT, TOKEN_NAME, 1, 'g', TOKEN_SEMICOLON,
TOKEN_FLOAT, TOKEN_NAME, 1, 'b', TOKEN_SEMICOLON,
TOKEN_CBRACE
```

The closed template *Matrix4x4* is identified by the UUID {55b6d781-37ec-11d0-ab39-0020af71e433} and has one member—a two-dimensional array named *matrix* of type *float*.

```
TOKEN_TEMPLATE, TOKEN_NAME, 9, 'M', 'a', 't', 'r', 'i', 'x', '4', 'x', '4', TOKEN_OBRACE,
TOKEN_GUID, 55b6d781, 37ec, 11d0, ab, 39, 00, 20, af, 71, e4, 33,
TOKEN_ARRAY, TOKEN_FLOAT, TOKEN_NAME, 6, 'm', 'a', 't', 'r', 'i', 'x',
TOKEN_OBRACKET, TOKEN_INTEGER, 4, TOKEN_CBRACKET,
TOKEN_OBRACKET, TOKEN_INTEGER, 4, TOKEN_CBRACKET,
TOKEN_CBRACE
```

The binary data object that follows shows an instance of the *RGB* template defined earlier. The example object is named *blue*, and its three members *r*, *g*, and *b* have the values 0.0, 0.0 and 1.0, respectively. Note that data is stored in little-endian format, which is not shown in this example.

```
TOKEN_NAME, 3, 'R', 'G', 'B', TOKEN_NAME, 4, 'b', 'l', 'u', 'e', TOKEN_OBRACE,
TOKEN_FLOAT_LIST, 3, 0.0, 0.0, 1.0, TOKEN_CBRACE
```

Direct3DX Shader Assemblers Reference

This section contains reference information for the vertex and pixel shader assemblers provided by the Direct3DX utility library.

- Vertex Shader Assembler Reference
- Pixel Shader Assembler Reference

All of the example syntax used in this documentation is intended to demonstrate how to write shaders when using the Direct3DX vertex and pixel shader assemblers.

Vertex Shader Assembler Reference

This section contains reference information for the vertex shader assembler. Reference material is divided into the following categories.

- Registers
- Instructions
- Modifiers

The vertex shader assembler is comprised of a set of registers defined along with a set of operations that can be performed on the registers. Operations are expressed as instructions comprised of an operator and one or more arguments (operands).

Registers

This section contains reference information for the input and output registers implemented by the Direct3DX vertex shader assembler. Information is divided into the following topics.

- Input Registers
- Output Registers

All registers are four-component floating-point vectors. Vector elements are designated as x, y, z and w in this documentation, but no semantics are implied.

Input Registers

The following table lists the minimum available input registers for use by a Microsoft® DirectX® 8.0 vertex shader. Note that this table applies to version 1.0 pixel shaders.

Name	Count	I/O Permissions	Number Allowed per Instruction
a*n*	1 scalar	write/use only	0 in version 1.0; 1 in version 1.1
c[*n*]	96 vectors	read-only	1
r*n*	12 vectors	read/write	3
v*n*	16 vectors	read-only	1

a*n*

These are the address registers. These registers are not available in vertex shader version 1.0 of Microsoft® DirectX® 8.0, and only a single register is available in version 1.1. The address register, designated as a0.x, may be used as signed integer offset in relative addressing into the constant register file.

```
c[a0.x + n]
```

Reads from out of the legal range will return (0.0, 0.0, 0.0, 0.0). Address registers may be a destination only for the **mov** instruction. Address registers cannot be read by the vertex shader, just used in relative addressing of the constant register file. Addresses read from outside the range of constant registers supported will return unpredictable results.

Version 1.0 vertex shaders that make any use of the address register will fail the Microsoft Direct3D® API call to create the vertex shader. Version 1.1 vertex shaders that attempt to use the address register before setting it will also fail.

c[*n*]

These are the constant registers. There are at least 96 four-component floating-point vectors comprising the constant register file. The constant registers are designated as either absolute or relative.

```
c[n]          ; absolute
c[a0.x + n]   ; relative - supported only in version 1.1
```

Therefore, the constant register may be read by using an absolute address, or addressed relative to an address register. Reads from out-of-range registers return (0.0, 0.0, 0.0, 0.0).

The constant register file is read-only from the perspective of the vertex shader. Any single instruction may access only one constant register. However, each source in that instruction may independently swizzle and negate that vector as it is read.

[C++]
The **MaxVertexShaderConst** member of **D3DCAPS8** indicates the number of available constant registers. Microsoft® DirectX® 8.0 shaders support at least 96 constants.

The constant register file has its data loaded by calling the Microsoft Direct3D® API function to set the vertex shader constant register. Alternatively, when creating a shader, the user can specify what constants should be loaded for a C++ application by an **IDirect3DDevice8::SetVertexShader** call.

[Visual Basic]
The **MaxVertexShaderConst** member of **D3DCAPS8** indicates the number of available constant registers. Microsoft® DirectX® 8.0 shaders support at least 96 constants.

The constant register file has its data loaded by calling the Microsoft Direct3D® API function to set the vertex shader constant register. Alternatively, when creating a shader, the user can specify what constants should be loaded for a Visual Basic application by an **Direct3DDevice8.SetVertexShader** call.

---

r*n*

These are the temporary registers. The temporary registers are grouped into a file of 12 4-D floating point vectors. Each temporary register has single-write and triple-read access. Therefore, an instruction can have as many as three temporary registers in its set of input source operands.

No values in a temporary register that remain from preceding invocations of the vertex shader can be used. Vertex shaders that read a value from a temporary register

before writing to it will fail the Microsoft® Direct3D® API call to create the vertex shader.

vn

These are the input vertex registers. Each vertex from one or more input vertex streams is loaded into the vertex input registers before the vertex shader is executed. The vertex input registers are a register file consisting of 16 four-component floating-point vectors, designated as v0 through v15. These registers are read only.

Any single instruction may access only one vertex input register. However, each source in the instruction may independently swizzle and negate that vector as it is read.

A declaration passed to the Microsoft® Direct3D® API call to create the vertex shader defines the mapping between the source vertex stream and the vertex input registers. For the fixed function pipeline, the input registers have the fixed mapping shown in the following table.

Vector Component	**Register**
Position	0
Blend Weight	1
Blend Indices	2
Normal	3
Point Size	4
Diffuse	5
Specular	6
Texture Coordinate 0	7
Texture Coordinate 1	8
Texture Coordinate 2	9
Texture Coordinate 3	10
Texture Coordinate 4	11
Texture Coordinate 5	12
Texture Coordinate 6	13
Texture Coordinate 7	14
Position 2	15
Normal 2	16

Output Registers

Output registers are defined as the inputs to the rasterizer. Generated data is written into the output register file, which has write-only access.

The output registers are defined by the register model of the raster shader, which must support the following minimum counts.

Name	I/O Permissions	Count
oD*n*	write-only	2 4-D vectors
oFog	write-only	1 scalar float
oPos	write-only	1 4-D vector
oPts	write-only	1 scalar float
oT*n*	write-only	4 4-D vectors

Write operations to destination registers not indicated above are ignored.

Register names are preceded by a lower case 'o,' indicating that the output registers are write-only.

When using vertex shaders with an implementation supporting only the Microsoft® DirectX® 6.0 or 7.0 pixel processing model—that is, an implementation without pixels shaders—the following output registers are available.

Name	I/0 Permissions	Count	Description
oD*n*	write-only	2 vectors	Output color data. Required for diffuse color
oPos	write-only	1 vector	Output position. Required for clipping.
oT*n*	write-only	2 vectors	Output texture coordinates. Required for max simultaneously bound to the texture blending

oD*n*

The output data registers, which are used to output vertex color data. Specifically, these are an array of output data registers that are iterated and directly routed to the pixel shader. Typically, these registers are used for colors.

oFog

The output fog value registers. The value is the fog factor to be interpolated and then routed to the fog table. Only the scalar x-component of the fog is used.

oPos

The output position registers. The value is the position in homogeneous clipping space. This value must be written by the vertex shader.

oPts

The output point-size registers. Only the scalar x-component of the point size is used.

oT*n*

The output texture coordinates registers. Specifically, these are an array of output data registers that are iterated and used as texture coordinate by the sampling stages routing data to the pixel shader.

Instructions

This section contains reference information for the instructions implemented by the Direct3DX vertex shader assembler. Information is divided into the following topics.

- General Instructions
- Version and Constant Definition Instructions
- Macro Instructions
- Modifiers

The vertex shader program can consist of up to 128 instructions, and instructions can have a maximum of three input arguments (source register operands).

General Instructions

The Direct3DX vertex shader assembler supports the following general instructions.

add	Sum
dp3	Three-components dot-product
dp4	Four-component dot-product
dst	Distance vector
expp	Exponential 10-bit precision
lit	Lighting coefficients
logp	Logarithm 10-bit precision
mad	Multiply and add
max	Maximum
min	Minimum
mov	Move
mul	Multiply
rcp	Reciprocal

rsq	Reciprocal square root
sge	Set on greater or equal than
slt	Set on less than
sub	Subtract

Each source operand may be arbitrarily swizzled on read, including broadcast/replicate. Each source operand may be negated.

Writes to the destination registers can include masking of individual components; that is, only the specified components (x,y,z,or w) are updated. No swizzling or negation is supported on writes. Therefore, the output masks must be in the x, y, z, and w order.

The software emulated front end will use the masks on the output register writes to detect what parts of the output vertex a shader modifies and compute an output flexible vertex format (FVF) accordingly for submission to the hardware for rasterization-only use.

Note that unlike the pixel shader language, no linear interpolation blend instruction is supported in the vertex shader language, and that the **dp3** operator does not automatically clamp negative results to 0.0.

See Also

Macro Instructions

add

Adds sources.

```
add   vDest, vSrc0, vSrc1
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **add** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);

m_TmpReg.x = m_Source[0].x + m_Source[1].x;
m_TmpReg.y = m_Source[0].y + m_Source[1].y;
m_TmpReg.z = m_Source[0].z + m_Source[1].z;
m_TmpReg.w = m_Source[0].w + m_Source[1].w;

WriteResult();
```

Remarks

The following examples illustrate how the **add** instruction might be used.

```
add r0, v0,c2
add r1.xyz, r0.xyz, v1.xyz
```

dp3

Computes the three-component dot product of the sources.

```
dp3   vDest, vSrc0, vSrc1
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **dp3** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);

m_TmpReg.x =
m_TmpReg.y =
m_TmpReg.z =
```

```
m_TmpReg.w = m_Source[0].x * m_Source[1].x +
        m_Source[0].y * m_Source[1].y +
        m_Source[0].z * m_Source[1].z;

WriteResult();
```

Remarks

The following example illustrates how the **dp3** instruction might be used.

```
dp3 r2, r0, r1
dp3 r2.x, r0, r1
```

dp4

Computes the four-component dot product of the sources.

```
dp4   vDest, vSrc0, vSrc1
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **dp4** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);

m_TmpReg.x =
m_TmpReg.y =
m_TmpReg.z =
m_TmpReg.w = m_Source[0].x * m_Source[1].x +
        m_Source[0].y * m_Source[1].y +
        m_Source[0].z * m_Source[1].z +
        m_Source[0].w * m_Source[1].w;
```

```
WriteResult();
```

Remarks

The following example illustrates how the **dp4** instruction might be used.

```
dp4 r2, r0, r1
dp4 r2.w, r0, r1
```

dst

Calculates the distance vector.

```
dst   vDest, vSrc0, vSrc1
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **dst** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);

m_TmpReg.x = 1;
m_TmpReg.y = m_Source[0].y * m_Source[1].y;
m_TmpReg.z = m_Source[0].z;
m_TmpReg.w = m_Source[1].w;

WriteResult();
```

Remarks

The first source operand is assumed to be the vector (*ignored*, d*d, d*d, *ignored*) and the second source operand is assumed to be the vector (*ignored*, 1/d, *ignored*, 1/d). The destination is the result vector (1, d, d*d, 1/d).

The following example illustrates how the **dst** instruction might be used.

```
dst r4, r0, r1
```

expp

Provides exponential 2^x partial support.

```
expp   vDest, vSrc0
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **expp** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);

float w = m_Source[0].w;
float v = (float)floor(m_Source[0].w);

m_TmpReg.x = (float)pow(2, v);
m_TmpReg.y = w - v;
// Reduced precision exponent
float tmp = (float)pow(2, w);
DWORD tmpd = *(DWORD*)&tmp & 0xffffff00;
m_TmpReg.z = *(float*)&tmpd;
m_TmpReg.w = 1;

WriteResult();
```

Remarks

The **expp** instruction produces undefined results if fed a negative value for the exponent.

This instruction provides exponential base 2 partial precision. It generates an approximate answer in **vDest**.z and allows for a more accurate determination of

vDest.x*function(**vDest**.y), where function is a user approximation to 2***vDest**.y over the limited range (0.0 <= **vDest**.y < 1.0).

This instruction accepts a scalar source, and reduced precision arithmetic is acceptable in evaluating **vDest**.z. However, the approximation error must be less than $1/(2^{11})$ the absolute error (10-bit precision) and over the range (0.0 <= t.y < 1.0). Also, **expp** returns 1.0 in w.

The following example illustrates how the **expp** instruction might be used.

```
expp r5, r0
```

lit

Provides lighting partial support.

```
lit   vDest, vSrc0
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **lit** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);

m_TmpReg.x = 1;
m_TmpReg.y = 0;
m_TmpReg.z = 0;
m_TmpReg.w = 1;
float power = m_Source[0].w;
const float MAXPOWER = 127.9961f;
if (power < -MAXPOWER)
   power = -MAXPOWER;        // Fits into 8.8 fixed point format
else
if (power > MAXPOWER)
   power = -MAXPOWER;        // Fits into 8.8 fixed point format

if (m_Source[0].x > 0)
{
```

```
    m_TmpReg.y = m_Source[0].x;
    if (m_Source[0].y > 0)
    {
       // Allowed approximation is EXP(power * LOG(m_Source[0].y))
       m_TmpReg.z = (float)(pow(m_Source[0].y, power));
    }
}

WriteResult();
```

Remarks

The **lit** instruction produces undefined results if fed a negative value for the exponent.

This instruction calculates lighting coefficients from two dot products and a power. The source vector is assumed to contain the values shown in the following pseudocode.

```
vSrc0.x = N*L       ; The dot product between normal and direction to light.
vSrc0.y = N*H       ; The dot product between normal and half vector.
vSrc0.z = ignored   ; This value is ignored.
vSrc0.w = power     ; The power, this value must be in the range from –128.0 through 128.0.
```

Reduced precision arithmetic is acceptable in evaluating **vDest**.z. An implementation must support at least 8 fraction bits in the power argument. Dot products are calculated with normalized vectors, and clamp limits are -128 to 128.

Error should correspond to a **logp** and **expp** combination, or no more than approximately 1 least significant bit (LSB) for an 8-bit color component.

The following example illustrates how the **lit** instruction might be used.

```
lit r0, r1
```

logp

Provides $\log_2(x)$ partial support.

```
logp   vDest, vSrc0
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **logp** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);

float v = ABSF(m_Source[0].w);
if (v != 0)
{
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w =
      (float)(log(v)/log(2));
}
else
{
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w = MINUS_INFINITY();
}

WriteResult();
```

Remarks

This instruction provides logarithm base 2 partial precision. It generates an approximate answer in **vDest**.z and allows for a more accurate determination of **vDest**.x + function(**vDest**.y), where function is a user approximation to $\log_2$(**vDest**.y) over the limited range (1.0 <= **vDest**.y < 2.0).

The **logp** instruction accepts a scalar source of which the sign bit is ignored, and reduced precision arithmetic is acceptable in evaluating **vDest**.z. The approximation error must be less than $1/(2^{11})$ in absolute error (10-bit precision), and over the range (1.0 <= t.y < 2.0). A zero source argument generates the result vector (-infinity, 0.0, -infinity, 1.0).

The following example illustrates how the **logp** instruction might be used.

```
logp r0,r1
```

mad

Multiplies and adds sources.

```
mad   vDest, vSrc0, vSrc1, vSrc2
```

Registers

vDest

Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

vSrc2
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **mad** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);
SetSrcReg(2);

m_TmpReg.x = m_Source[0].x * m_Source[1].x + m_Source[2].x;
m_TmpReg.y = m_Source[0].y * m_Source[1].y + m_Source[2].y;
m_TmpReg.z = m_Source[0].z * m_Source[1].z + m_Source[2].z;
m_TmpReg.w = m_Source[0].w * m_Source[1].w + m_Source[2].w;

WriteResult();
```

Remarks

The following examples illustrate how the **mad** instruction might be used.

```
mad r0,r1,r2,v3
```

max

Calculates the maximum of the sources.

```
max   vDest, vSrc0, vSrc1
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **max** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);

m_TmpReg.x=(m_Source[0].x >= m_Source[1].x) ? m_Source[0].x : m_Source[1].x;
m_TmpReg.y=(m_Source[0].y >= m_Source[1].y) ? m_Source[0].y : m_Source[1].y;
m_TmpReg.z=(m_Source[0].z >= m_Source[1].z) ? m_Source[0].z : m_Source[1].z;
m_TmpReg.w=(m_Source[0].w >= m_Source[1].w) ? m_Source[0].w : m_Source[1].w;

WriteResult();
```

Remarks

The following example illustrates how the **max** instruction might be used.

```
max r2, r3, r4
```

min

Calculates the minimum of the sources.

```
min   vDest, vSrc0, vSrc1
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **min** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);
```

```
m_TmpReg.x=(m_Source[0].x < m_Source[1].x) ? m_Source[0].x : m_Source[1].x;
m_TmpReg.y=(m_Source[0].y < m_Source[1].y) ? m_Source[0].y : m_Source[1].y;
m_TmpReg.z=(m_Source[0].z < m_Source[1].z) ? m_Source[0].z : m_Source[1].z;
m_TmpReg.w=(m_Source[0].w < m_Source[1].w) ? m_Source[0].w : m_Source[1].w;

WriteResult();
```

Remarks

The following example illustrates how the **min** instruction might be used.

```
min   r2, r3, r4
```

mov

Move contents of the source into the destination.

```
mov   vDest, vSrc0
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **mov** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);

if( m_pDest == m_reg.m_a )
{
   float p = (float)floor(m_Source[0].x);
   *(int*)&m_pDest->x = FTOI(p);
}
else
{
   m_TmpReg = m_Source[0];
   WriteResult();
}
```

Remarks

The following examples illustrate how the **mov** instruction might be used.

```
mov r1, v2
mov r1.xy, r2.zw
```

mul

Multiply sources into the destination.

```
mul   vDest, vSrc0, vSrc1
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **mul** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);

m_TmpReg.x = m_Source[0].x * m_Source[1].x;
m_TmpReg.y = m_Source[0].y * m_Source[1].y;
m_TmpReg.z = m_Source[0].z * m_Source[1].z;
m_TmpReg.w = m_Source[0].w * m_Source[1].w;

WriteResult();
```

Remarks

The following example illustrates how the **mul** instruction might be used.

```
mul r4, v0, r1
```

rcp

Computes the reciprocal of the source scalar.

```
rcp   vDest, vSrc0
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **rcp** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);

if( m_Source[0].w == 1.0f )
{
   // Must be exactly 1.0
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w = 1.0f;
}
else if( m_Source[0].w == 0 )
{
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w = PLUS_INFINITY();
}
else
{
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w = 1.0f/m_Source[0].w;
}

WriteResult();
```

Remarks

If the source has no subscripts, the x-component is used. The output must be exactly 1.0 if the input is exactly 1.0.

Precision should be at least $1.0/(2^{22})$ absolute error over the range (1.0, 2.0) because common implementations will separate mantissa and exponent. A source of 0.0 yields infinity.

The following example illustrates how the **rcp** instruction might be used.

```
rcp r1, r2
rcp r1, r2.y
```

rsq

Computes the reciprocal square root of the source scalar.

```
rsq   vDest, vSrc0
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **rsq** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);

float v = ABSF(m_Source[0].w);
if( v == 1.0f )
{
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w = 1.0f;
}
else if( v == 0 )
{
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w = PLUS_INFINITY();
}
else
{
   v = (float)(1.0f / sqrt(v));
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w = v;
}

WriteResult();
```

Remarks

If source has no subscripts, the x-component is used. The output must be exactly 1.0 if the input is exactly 1.0.

Absolute value is taken before processing; that is, the sign bit is ignored.

Precision should be at least $1.0/(2^{22})$ absolute error over the range (1.0, 4.0) because common implementations will separate mantissa and exponent. A source of 0.0 yields infinity.

The following example illustrates how the **rsq** instruction might be used.

```
rsq r1, r2
rsq r1, r2.y
```

sge

Sets the destination to 1.0 if the first source operand is greater than or equal to the second source operand; otherwise, the destination is set to 0.0.

```
sge   vDest, vSrc0, vSrc1
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **sge** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);

m_TmpReg.x = (m_Source[0].x >= m_Source[1].x) ? 1.0f : 0.0f;
m_TmpReg.y = (m_Source[0].y >= m_Source[1].y) ? 1.0f : 0.0f;
m_TmpReg.z = (m_Source[0].z >= m_Source[1].z) ? 1.0f : 0.0f;
m_TmpReg.w = (m_Source[0].w >= m_Source[1].w) ? 1.0f : 0.0f;

WriteResult();
```

Remarks

The following example illustrates how the **sge** instruction might be used.

```
sge   r1, r2, v1
```

slt

Sets the destination to 1.0 if the first source operand is less than the second source operand; otherwise, the destination is set to 0.0.

```
slt   vDest, vSrc0, vSrc1
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

vSrc1
Source register, specifying the input argument.

Operation

The following code fragment shows the operations performed by the **slt** instruction to write a result to the destination.

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1);

m_TmpReg.x = (m_Source[0].x < m_Source[1].x) ? 1.0f : 0.0f;
m_TmpReg.y = (m_Source[0].y < m_Source[1].y) ? 1.0f : 0.0f;
m_TmpReg.z = (m_Source[0].z < m_Source[1].z) ? 1.0f : 0.0f;
m_TmpReg.w = (m_Source[0].w < m_Source[1].w) ? 1.0f : 0.0f;

WriteResult();
```

Remarks

The following example illustrates how the **slt** instruction might be used.

```
slt   r1, r2, v1
```

sub

Subtracts sources.

```
sub   tDest, tSrc0, tSrc1
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

tSrc1
Source register, specifying the input argument.

Remarks

This instruction performs the subtraction based on the following formula.

```
tDest = tSrc0 - tSrc1
```

The following example shows how this instruction might be used.

```
sub   r2, v2, r2    ; difference
```

Version and Constant Definition Instructions

The Direct3DX vertex shader assembler provides instructions to identify the type and version of the shader and to define constants to use with the vertex shader.

def	Define Constants
vs	Version and Type

A version instruction is required to be at the beginning of each shader.

Note that constant definition instructions must occur after version instructions and before all other instructions.

def

Provides a method to define constants to be used with the vertex shader.

```
def   vDest, fVal0, fVal1, fVal2, fVal3
```

Registers

vDest
Destination register, holding the result of the operation.

fVal0
Source register, specifying the input argument.

fVal1
Source register, specifying the input argument.

fVal2
Source register, specifying the input argument.

fVal3
Source register, specifying the input argument.

Remarks

The following example illustrates how the **def** instruction might be used.

```
def c0, 0.0f, 0.0f, 0.0f, 1.0f
def c8, 1.0f, 2.0f, 3.0f, 4.0f
```

These constants are returned as a declaration fragment to be included in the declaration upon shader creation.

VS

Provides a method to specify the type and version of the shader code.

```
vs.mainVer.subVer
```

Registers

mainVer
Main version number of the vertex shader.

subVer
Sub version number of the vertex shader.

Remarks

The following example illustrates how the **vs** instruction might be used.

```
vs.1.0  ; Vertex shader version 1.0.
```

This instruction is required to be at the beginning of all vertex shaders.

Macro Instructions

The Direct3DX vertex shader assembler provides macro instructions for application convenience in performing common simple operations. These macros are expanded by the driver into some of the standard vertex shader instructions, for details see the reference topic for each macro instruction.

exp	Exponential base 2 full precision
frc	Fraction

log	Logarithm base 2 full precision
m3x2	3×2 vector matrix multiply
m3x3	3×3 vector matrix multiply
m3x4	3×4 vector matrix multiply
m4x3	4×3 vector matrix multiply
m4x4	4×4 vector matrix multiply

For the purpose of determining if you exceed the maximum instruction count limit of 128, the macro instructions are guaranteed to expand to no more than the number of actual instructions listed.

Because these are passed to the implementation in the same manner as true instructions, the implementation is free to optimize them. Therefore, no assumptions can be made about timing cycle counts required.

See Also

General Instructions

exp

Provides exponential 2^x with full precision to at least $1/2^{20}$.

```
exp   vDest, vSrc0
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

Operation

```
SetDestReg();
SetSrcReg(0);

float v = m_Source[0].w;

m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w = (float)pow(2, v);

WriteResult();
```

Expansion

This macro takes twelve instruction slots.

Remarks

This is a scalar operation and takes its input from the .w channel, which must be specified. It always replicates the result into all four output channels.

frc

Returns fractional portion of each input component.

```
frc   vDest, vSrc0
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

Operation

```
SetDestReg();
SetSrcReg(0);

m_TmpReg.x = m_Source[0].x - (float)floor(m_Source[0].x);
m_TmpReg.y = m_Source[0].y - (float)floor(m_Source[0].y);
m_TmpReg.z = m_Source[0].z - (float)floor(m_Source[0].z);
m_TmpReg.w = m_Source[0].w - (float)floor(m_Source[0].w);

WriteResult();
```

Expansion

This macro takes three instruction slots.

Remarks

Each component of the result is in the range from 0.0 through 1.0.

This macro only writes x and y components.

log

Provides $\log_2(x)$ support with full float precision of at least $1/2^{20}$.

```
log   vDest, vSrc0
```

Registers

vDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

Operation

```
SetDestReg();
SetSrcReg(0);

float v = ABSF(m_Source[0].w);
if (v != 0)
{
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w =
      (float)(log(v)/log(2));
}
else
{
   m_TmpReg.x = m_TmpReg.y = m_TmpReg.z = m_TmpReg.w = minUS_INFINITY();
}

WriteResult();
```

Expansion

This macro takes twelve instruction slots.

Remarks

This instruction accepts a scalar source .w of which the sign bit is ignored. The result is replicated to all four channels.

The input exponent must be in the range –128 to 128. The approximation error must be less than $1/2^{20}$ in absolute error, and over the range 1.0 less than or equal to t.y less than 2.0. A zero source generates (-infinity,-infinity,-infinity,-infinity).

m3x2

Computes the product of the input vector and a 3x2 matrix.

```
m3x2   rDest, vSrc0, mSrc1
```

Registers

rDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

mSrc1
Source register, specifying the input argument.

Operation

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1, 2);
m_TmpReg.x = m_Source[0].x * m_Source[1].x +
        m_Source[0].y * m_Source[1].y +
        m_Source[0].z * m_Source[1].z;
m_TmpReg.y = m_Source[0].x * m_Source[2].x +
        m_Source[0].y * m_Source[2].y +
        m_Source[0].z * m_Source[2].z;

WriteResult();
```

Expansion

The following **m3x2** instruction

```
m3x2   r5, v0, c[3]
```

expands to

```
dp3   r5.x, v0, c[3]
dp3   r5.y, v0, c[4]
```

Therefore, this macro instruction consumes two instruction slots from the instruction count. Note that the last w-component in c[3] and c[4] is ignored in this computation.

Remarks

The input vector is at **vSrc0**, the input 3x2 matrix is at **mSrc1**, and the next higher register in the same register file. A 2-D result is produced, leaving the other elements of **rDest** (z and w) unaffected.

This operation is commonly used for 2-D transforms. This macro instruction is implemented as a pair of dot products.

m3x3

Computes the product of the input vector and a 3x3 matrix.

```
m3x3   rDest, vSrc0, mSrc1
```

Registers

rDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

mSrc1
Source register, specifying the input argument.

Operation

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1, 3);
m_TmpReg.x = m_Source[0].x * m_Source[1].x +
        m_Source[0].y * m_Source[1].y +
        m_Source[0].z * m_Source[1].z;
m_TmpReg.y = m_Source[0].x * m_Source[2].x +
        m_Source[0].y * m_Source[2].y +
        m_Source[0].z * m_Source[2].z;
m_TmpReg.z = m_Source[0].x * m_Source[3].x +
        m_Source[0].y * m_Source[3].y +
        m_Source[0].z * m_Source[3].z;

WriteResult();
```

Expansion

The following **m3x3** instruction

```
m3x3   r5, v0, c[3]
```

expands to

```
dp3   r5.x, v0, c[3]
dp3   r5.y, v0, c[4]
dp3   r5.z, v0, c[5]
```

Therefore, this macro instruction consumes three instruction slots from the instruction count. Note that the last w-component in c[3], c[4], and c[5] is ignored in this computation.

Remarks

The input vector is at **vSrc0** and the input 3x3 matrix is at **mSrc1** and the two subsequent registers in the same register file.

This operation is commonly used for transforming normal vectors during lighting computations. This macro instruction is implemented as a series of dot products.

m3x4

Computes the product of the input vector and a 3x4 matrix.

```
m3x4   rDest, vSrc0, mSrc1
```

Registers

rDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

mSrc1
Source register, specifying the input argument.

Operation

```
SetDestReg();
SetSrcReg(0);
SetSrcReg(1, 4);
m_TmpReg.x = m_Source[0].x * m_Source[1].x +
        m_Source[0].y * m_Source[1].y +
        m_Source[0].z * m_Source[1].z;
m_TmpReg.y = m_Source[0].x * m_Source[2].x +
        m_Source[0].y * m_Source[2].y +
        m_Source[0].z * m_Source[2].z;
m_TmpReg.z = m_Source[0].x * m_Source[3].x +
        m_Source[0].y * m_Source[3].y +
        m_Source[0].z * m_Source[3].z;
m_TmpReg.w = m_Source[0].x * m_Source[4].x +
        m_Source[0].y * m_Source[4].y +
        m_Source[0].z * m_Source[4].z;

WriteResult();
```

Expansion

The following **m3x4** instruction

```
m3x4   r5, v[0], c[3]
```

expands to

```
dp3 r5.x, v[0], c[3]
dp3 r5.y, v[0], c[4]
dp3 r5.z, v[0], c[5]
dp3 r5.w, v[0], c[6]
```

Therefore, this macro instruction consumes four instruction slots from the instruction count. Note that the last w-component in c[3], c[4], and c[5] is ignored in this computation.

Remarks

The input vector is at **vSrc0** and the input 3x4 matrix is at **mSrc1** and the two subsequent registers in the same register file.

This operation is commonly used for transforming a position vector by a matrix that has a projective effect, but applies no translation. This macro instruction is implemented as a series of dot products.

m4x3

Computes the product of the input vector and a 4x3 matrix.

```
m4x3   rDest, vSrc0, mSrc1
```

Registers

rDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

mSrc1
Source register, specifying the input argument.

Operation

```
SetDestReg();
SetSrcReg(0);
```

```
SetSrcReg(1, 3);
m_TmpReg.x = m_Source[0].x * m_Source[1].x +
        m_Source[0].y * m_Source[1].y +
        m_Source[0].z * m_Source[1].z +
        m_Source[0].w * m_Source[1].w;
m_TmpReg.y = m_Source[0].x * m_Source[2].x +
        m_Source[0].y * m_Source[2].y +
        m_Source[0].z * m_Source[2].z +
        m_Source[0].w * m_Source[2].w;
m_TmpReg.z = m_Source[0].x * m_Source[3].x +
        m_Source[0].y * m_Source[3].y +
        m_Source[0].z * m_Source[3].z +
        m_Source[0].w * m_Source[3].w;

WriteResult();
```

Expansion

The following **m4x3** instruction

```
m4x3   r5, v[0], c[3]
```

expands to

```
dp4   r5.x, v[0], c[3]
dp4   r5.y, v[0], c[4]
dp4   r5.z, v[0], c[5]
```

Therefore, this macro instruction consumes three instruction slots from the instruction count. Note that the last w-component in c[3], c[4], and c[5] is ignored in this computation unless the input vector has a w-value of 1.0. If this w-value is 0.0, then no translation of the input vector will occur; that is, the translation elements of the matrix will not be applied.

Remarks

The input vector is at **vSrc0** and the input 4x3 matrix is at **mSrc1** and the two next higher registers in the same register file.

This operation is commonly used for transforming a position vector by a matrix that has no projective effect, such as occurs in model-space transformations. This macro instruction is implemented as a series of dot products.

m4x4

Computes the product of the input vector and a 4x4 matrix.

```
m4x4   rDest, vSrc0, mSrc1
```

Registers

rDest
Destination register, holding the result of the operation.

vSrc0
Source register, specifying the input argument.

mSrc1
Source register, specifying the input argument.

Operation

```
SetDestReg();
 SetSrcReg(0);
 SetSrcReg(1, 4);
 m_TmpReg.x = m_Source[0].x * m_Source[1].x +
          m_Source[0].y * m_Source[1].y +
          m_Source[0].z * m_Source[1].z +
          m_Source[0].w * m_Source[1].w;
 m_TmpReg.y = m_Source[0].x * m_Source[2].x +
          m_Source[0].y * m_Source[2].y +
          m_Source[0].z * m_Source[2].z +
          m_Source[0].w * m_Source[2].w;
 m_TmpReg.z = m_Source[0].x * m_Source[3].x +
          m_Source[0].y * m_Source[3].y +
          m_Source[0].z * m_Source[3].z +
          m_Source[0].w * m_Source[3].w;
 m_TmpReg.w = m_Source[0].x * m_Source[4].x +
          m_Source[0].y * m_Source[4].y +
          m_Source[0].z * m_Source[4].z +
          m_Source[0].w * m_Source[4].w;

 WriteResult();
```

Expansion

The following **m4x4** instruction

```
m4x4   r5, v[0], c[3]
```

expands to

```
dp4   r5.x, v[0], c[3]
dp4   r5.y, v[0], c[4]
dp4   r5.z, v[0], c[5]
dp4   r5.w, v[0], c[6]
```

Therefore, this macro consumes four instruction slots from the instruction count.

Remarks

The input vector is at **vSrc0** and the input 4x4 matrix is at **mSrc1** and the three subsequent registers in the same file.

This operation is commonly used to transform an position by a projection matrix. This macro instruction is implemented as a series of dot products.

Modifiers

This section contains reference information for the component modifiers implemented by the Direct3DX vertex shader assembler.

As shown in the following table, component modifiers (where *r* is a valid register, either **a*n***, **c[*n*]**, **r*n***, **v*n***, or any of the output registers) can be applied to the individual components of the vector data.

Component Modifier	Description
r.{x}{y}{z}{w}	Destination mask
r.[xyzw][xyzw][xyzw][xywz]	Source swizzle
-r	Source negation

Pixel Shader Assembler Reference

This section contains reference information for the Direct3DX pixel shader assembler. Reference material is divided into the following categories.

- Registers
- Instructions
- Modifiers
- Output Write Masks

The pixel shader assembler is comprised of a set of registers defined along with a set of operations that can be performed on the registers. Operations are expressed as instructions comprised of an operator and one or more arguments (operands).

Microsoft® DirectX® 8.0 requires intermediate computations to maintain at least 8-bit precision for all surface formats. Both higher precision (12-bit) for in-stage math and saturation to 8-bits between texture stages are recommended. No modifiable rounding modes or exceptions are supported. Multiplication should be supported with a round-to-nearest precision to minimize precision loss.

Registers

This section contains reference information for the registers implemented by the Direct3DX pixel shader assembler.

Name	I/0 Permissions	Minimum Count	Maximum Count per Instruction	Source
c*n*	read-only	8	2	API call
r*n*	read/write	2	2	Written
t*n*	read/write	4	1	Textures
v*n*	read-only	2	1	Vertex Colors

The minimum count column indicates the minimum number of registers that the implementation must support in order to be able to expose Microsoft® DirectX® 8.0 shading capability.

The maximum count per instruction column indicates the maximum number of registers supported in any one instruction. This includes source parameter read ports for an instruction. The destination parameter of an instruction does not apply towards the limit.

Each register is a four-component vector, whose components range from -1.0 to 1.0, with at least 8 bits of precision per component. Unlike the vertex shader assembler, the pixel shader assembler does not have any output registers; rather, the pixel shader just emits r0.

c*n*

[C++]
These are the constant registers. Derived from the D3DRS_TEXTUREFACTOR constant, the constant factor registers are loaded by the application through the **IDirect3DDevice8::SetPixelShaderConstant** method.

[Visual Basic]
These are the constant registers. Derived from the D3DRS_TEXTUREFACTOR constant, the constant factor registers are loaded by the application through the **Direct3DDevice8.SetPixelShaderConstant** method.

The constant factor registers are read-only. There are at least eight constant factor registers defined for use by a pixel shader, but a maximum of two may be used in any one instruction.

r*n*

[C++]
These are the temporary registers. The temporary registers are available for use in storing intermediate results. They are read-write and a maximum of two temporary registers can appear in any single shader instruction. Shader preprocessing in the debug runtime will fail **IDirect3DDevice8::CreatePixelShader** on any shader that attempts to read from a temporary register that has not been written to by a previous instruction.

[Visual Basic]
These are the temporary registers. The temporary registers are available for use in storing intermediate results. They are read-write and a maximum of two temporary registers can appear in any single shader instruction. Shader preprocessing in the debug runtime will fail **Direct3DDevice8.CreatePixelShader** on any shader that attempts to read from a temporary register that has not been written to by a previous instruction.

Note that all pixel shaders must write **r0** as the final result, so the simplest decal pixel shader consists of two instructions.

```
ps.1.0      // DirectX8 Version.
tex   t0
mov   r0, t0
```

t*n*

[C++]
These are the texture registers. There are as many texture registers as there are simultaneous textures supported. The texture registers are initialized to contain texture colors from the texture sampling units including filtering modes defined by that stage's texture stage state. The colors come from their corresponding textures as defined by **IDirect3DDevice8::SetTexture** and sampled at the corresponding texture coordinates.

[Visual Basic]
These are the texture registers. There are as many texture registers as there are simultaneous textures supported. The texture registers are initialized to contain texture colors from the texture sampling units including filtering modes defined by that stage's texture stage state. The colors come from their corresponding textures as defined by **Direct3DDevice8.SetTexture** and sampled at the corresponding texture coordinates.

Any texture register that does not have a texture set will be sampled as opaque black (0,0,0,1). A maximum of two texture registers can appear in any single shader instruction. Texture registers may also be used as temporary registers. Specifically, after the texture addressing instructions, these registers can be used as read/write temporary registers.

In most cases, a texture register is only valid as a destination of a texture command; however, note that the arithmetic instructions can read from texture registers.

v*n*

These are the vertex color registers. The input vertex color registers contain color values obtained by per-pixel Gouraud iteration of the color values emitted by the vertex shader (which may in turn receive them from the vertex stream). A compliant Microsoft® DirectX® 8.0 implementation must support at least two of the input color registers.

Input color registers are read-only, and a maximum of one of these colors can be used in any one instruction.

To simulate flat shading, a constant color is more efficient. However, when the shade mode is D3DSHADE_FLAT, applications iteration of both colors is disabled. Note that fog should continue to be iterated. Ideally, all color iteration should be perspective correct in the same manner as the texture coordinate iteration.

Instructions

There are two main categories of operations understood by the Direct3DX compiler for the pixel shader: pixel color and alpha blending operations and texture addressing operations. The assembler represents these operations using instructions. Pixel color and alpha blending operations modify color data, and texture addressing operations affect and process texture coordinate data. All of the pixel shader instructions are performed on a per-pixel basis—that is, they have no knowledge of other pixels in the pipeline.

- Color and Alpha Blending Instructions
- Version and Constant Definition Instructions
- Texture Addressing Instructions

The blending operations can be used for independent processing of RGB color and scalar alpha. In most cases, pixel blending (color or alpha) operations can support both a color and an alpha instruction in one slot.

Implementations must support programs of at least eight blending instructions and four addressing instructions to be compliant.

Texture addressing operations each consume one slot, but pixel blending (color and alpha) operations can be paired to enable both a color and a alpha instruction in a single slot. In Microsoft® DirectX® 8.0, the texture addressing instructions must precede the color blending instructions.

The syntax for texture address perturbation operation is included within the shader program itself in the form of texture declarations. These control how address perturbation operations are applied to the color values sampled from the textures during their pre-loading into the temporary register file for use in the body of the pixel shader.

Color and Alpha Blending Instructions

This section contains reference information for the color and alpha blending instructions implemented by the Direct3DX pixel shader assembler.

The color and alpha blending instructions are listed in the following table. All of these operations are performed component-wise.

Instruction	Description
add	Add
cnd	Conditional
dp3	Three-Component Vector Dot-Product
lrp	Linear Interpolation Blend
mad	Multiply and Add
mov	Copy Source Into Destination
mul	Modulate
sub	Loads the difference of the two colors in the source operands.

add

Loads the sum of the two colors in the source operands.

```
add   tDest, tSrc0, tSrc1
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

tSrc1

Source register, specifying the input argument.

Remarks

The following example shows how this instruction might be used.

```
add d, s1, s2    ; sum d = s1 + s2
```

cnd

Performs a conditional operation, comparing the value in r0.a with 0.5.

```
cnd   tDest, tSrc0, tSrc1, tSrc2
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

tSrc1
Source register, specifying the input argument.

tSrc2
Source register, specifying the input argument.

Remarks

This instruction can only be used to compare the value in r0.a with 0.5. If the alpha (r0.a) is greater than 0.5, then the first source is returned as the result; otherwise, the second source is returned.

The following example shows how this instruction might be used.

```
cnd d, r0.a, s1, s2    ; d = ( r0.a >0.5 ? s1 : s2 )
```

To compare two values you can use the following pixel shader code because a - (b - 0.5) = a - b + 0.5.

```
sub r0, v0,   v1_bias
cnd r0, r0.a, c0, c1
```

As mentioned above, r0.a is the only value that can be placed in the first source argument and the compare value must be "greater than 0.5."

dp3

Loads the three-component vector dot-product of the two colors in the source operand registers.

```
dp3   tDest, tSrc0, tSrc1
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

tSrc1
Source register, specifying the input argument.

Remarks

This instruction generates a scalar result, but replicates it to all specified output channels.

```
dp3 r, s1, s2    ; s1 dot s2 replicated to all channels
```

Note that the **dp3** instruction is a fundamentally a vector operation, and is therefore always performed in the vector pipeline. Using it with the Alpha Replicate input argument modifier (.a) input argument modifier just indicates that its result should be propagated to the alpha channel as well. In other words, **dp3** can be specified as an alpha operation only when it is also the color operation, and must also use the same arguments.

```
dp3 r0.rgb, t0, v0
dp3 r0.a,   t0, v0
```

Or equivalently,

```
dp3 r0, t0, v0
```

will cause the grayscale result of the **dp3** operation to be copied into both the color and alpha channels of r0.

A different instruction can be specified in the alpha channel of the result.

```
dp3 r0.rgb, t0, v0
mul r0.a,   t0, v0
```

The **dp3** instruction usually benefits from the Signed Scaling input argument modifier (**_bx2**) applied to its input arguments if they have not already been expanded to signed dynamic range. When used for lighting, the saturate instruction modifier (**_sat**)

instruction modifier is often used to clamp the negative values to black, as shown in the following example.

```
dp3_sat r0, t0_bx2, v0_bx2    ; t0 is bump, v0 is light direction
```

lrp

Linearly interpolates between the 2nd and 3rd source registers by a proportion specified in the 1st source register.

```
lrp   tDest, tSrc0, tSrc1, tSrc2
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

tSrc1
Source register, specifying the input argument.

tSrc2
Source register, specifying the input argument.

Remarks

This instruction performs the linear interpolation based on the following formula.

```
tSrc0 * tSrc1 + (1-tSrc0) * tSrc2 = tSrc2 + tSrc0 * (tSrc1 - tSrc2)
```

The following example shows how this instruction might be used.

```
lrp d, s0, s1, s2    ; d = s0*s1 + (1-s0)*s2 = s2 + s0*(s1-s2)
```

mad

Loads the first source register added to the product of the two colors in the last two source operand registers.

```
mad   tDest, tSrc0, tSrc1, tSrc2
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0

Source register, specifying the input argument.

tSrc1
Source register, specifying the input argument.

tSrc2
Source register, specifying the input argument.

Remarks

This instruction performs a multiply-accumulate operation. It takes the last two arguments, multiplies them together, and adds them to the remaining input/source argument, and places that into the result register.

This instruction performs the multiply-add based on the following formula.

```
tSrc0 + tSrc1 * tSrc2
```

The following example shows how this instruction might be used.

```
mad  d, s0, s1, s2    ; d = s0 + s1*s2
```

mov

Loads with the value in the source register.

```
mov   tDest, tSrc0
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

Remarks

The following example shows how this instruction might be used.

```
mov   d, s    ; copy s to d ( d = s )
```

mul

Loads the destination register with the component-wise product of the two colors in the source operand registers

```
mul   tDest, tSrc0, tSrc1
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

tSrc1
Source register, specifying the input argument.

Remarks

This instruction performs a modulate operation.

The following example shows how this instruction might be used.

```
mul  d, s1, s2    ; modulate d = s1*s2
```

sub

Loads the difference of the two colors in the source operands.

```
sub   tDest, tSrc0, tSrc1
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

tSrc1
Source register, specifying the input argument.

Remarks

This instruction performs the subtraction based on the following formula.

```
tDest = tSrc0 - tSrc1
```

The following example shows how this instruction might be used.

```
sub   d, s1, s2    ; difference
```

Version and Constant Definition Instructions

The Direct3DX pixel shader assembler provides instructions to identify the type and version of the shader and to define constants to use with the pixel shader.

def	Define Constants
ps	Version and Type

A version instruction is required to be at the beginning of each shader.

Note that constant definition instructions must occur after version instructions and before all other instructions.

def

Provides a method to define constants to be used with the pixel shader.

```
def   vDest, fVal0, fVal1, fVal2, fVal3
```

Registers

vDest
Destination register, holding the result of the operation.

fVal0
Source register, specifying the input argument.

fVal1
Source register, specifying the input argument.

fVal2
Source register, specifying the input argument.

fVal3
Source register, specifying the input argument.

Remarks

The following example illustrates how the **def** instruction might be used. Note that the **def** instruction must come before other shader instructions.

```
Ps.1.0
def c0, 0.0f, 0.0f, 0.0f, 1.0f
def c7, 1.0f, 2.0f, 3.0f, 4.0f
tex t0
mov r0, t0
```

These constants are defined in the instruction stream.

ps

Provides a method to specify the type and version of the shader code.

```
ps.mainVer.subVer
```

Registers

mainVer
Main version number of the pixel shader.

subVer
Sub version number of the pixel shader.

Remarks

The following example illustrates how the **ps** instruction might be used.

```
ps.1.0  ; Pixel shader version 1.0.
```

This instruction is required to be at the beginning of all pixel shaders.

Texture Addressing Instructions

This section contains reference information for the texture addressing instructions implemented by the Direct3DX pixel shader assembler. The texture addressing instructions are provided to enable manipulations of texture address data (as opposed to texture color contents).

The textures set at the sampling stages are sampled from the corresponding textures using the sampling, filtering, and wrapping modes selected by the Microsoft® Direct3D® application programming interface (API) texture stage states. When these textures are sampled, some operations may be applied to the address used to sample them. These operations are referred to as texture addressing operations.

A subset of the texture addressing instructions and the texture perturbation operations are listed in the following table.

Instruction	Description
tex	No Modification
texbem	Bump Environment Map
texbeml	Bump Environment Map with Luminance
texcoord	Texture Coordinate
texkill	Mask Out Pixel
texm3x2pad	Input to 3×2 Matrix Multiply
texm3x2tex	3×2 Matrix Multiply Result
texreg2ar	Remapping Alpha and Red Components
texreg2gb	Remapping Green and Blue Components

In addition, the pixel shader API supports a number of methods to perform a per-pixel 3×3 matrix multiply on the vector sampled from a texture. The 3×3 matrix is

comprised of the texture coordinates from three consecutive texture declaration stages. These matrix multiply operations are constructed from multiple texture address operations.

The following table lists the remaining texture addressing instructions, those that are concerned with per-pixel 3×3 matrix multiplies.

Instruction	Description
texm3x3pad	Input to 3×3 Matrix Multiply
texm3x3spec	Specular Reflection and Environment Mapping
texm3x3tex	3×3 Matrix Multiply Result
texm3x3vspec	Specular Reflection/Environment Mapping without Constant Eye Vector

Texture address instructions define an output argument (result) being the texture register that they declare/define. Some instructions require an input operand to be specified. This input operand must be a previously defined texture. These can also be thought of as declarations of the texture colors emitted by the texture sampling units into the color operation/alpha operation/blending shader. These texture registers can be declared in terms of previously declared textures, but not in terms of later textures, or other pixel color registers.

All texture address operations must be declared in the order of the texture they define. No texture address operation can refer to another texture that is after it —that is, of a higher stage number.

Performance Note

Texture addressing operations need to support sufficient precision for representing texture addresses. For example, precision should be sufficient enough to represent perturbations in the elements of the bump environment mapping matrix that can perturb to a fraction of a texel in an environment map of the maximum texture dimensions supported by the device. Furthermore, all iterated quantities in these operations must be iterated with perspective correction.

tex

Samples a color from the texture and copies it into the corresponding register with no modification applied to the addresses used for sampling.

```
tex  tDest
```

Registers

tDest

Destination register, holding the result of the operation.

Remarks

Because this instruction is the default, the **tex** instruction is assumed for any textures whose sampling is not specified by other statements.

The following example shows how this instruction might be used.

```
tex r0    ; Declare register 0 as color from the
          ; texture currently set at stage 0.
```

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texbem

Takes the color defined by the input argument as DuDv perturbation data.

```
texbem   tDest, tSrc0
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Temporary source register, specifying the input argument.

Remarks

The **texbem** instruction transforms the u and v components by the 2-D bump environment mapping matrix, adds them to the current stage's texture coordinates, and samples the current stage's texture.

```
tex    t0       ; Define t0 as being sampled from the texture.
texbem t1, t0   ; Compute u = u1 + mat00*r0.r + mat01*r0.g, and
               ;        v = v1 + mat10*r0.r + mat11*r0.g.
               ; Then sample at u,v; r0.b is ignored.
mov r0, t1      ; Copy final result to output color.
```

This operation always interprets du and dv as signed quantities, so that the Signed Scaling input modifier (**_bx2)** is not required to be specified on the input argument. The texture is set at the second stage, the texture set on the first stage is ignored.

This can be used for a variety of techniques based on address perturbation, including per-pixel environment mapping and diffuse lighting (bump mapping), environment matting, and so on.

This operation is a macro of two instructions (consuming 2 stages/slots): u' = u + du∗M00 + dv∗M01 and v' = v + du∗M10 + dv∗M11 where M00, M01, M10, and

M11 are D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT01, D3DTSS_BUMPENVMAT10, and D3DTSS_BUMPENVMAT11.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texbeml

Takes the color sampled by the preceding stage as DuDvL perturbation data with luminance information.

```
texbeml   tDest, tSrc0
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Temporary source register, specifying the input argument.

Remarks

The **texbeml** instruction transforms the color's red and green components by the 2-D bump environment mapping matrix, adds the components to the current stage's texture coordinates, and samples this texture at the resulting 2-D address. Then it applies a luminance correction using the luminance value (L), and the Microsoft® Direct3D® API-specified luminance and bias texture stage values.

The following pixel shader defines the color in the destination register as the color taken from the texture map set at stage 1, indexed by an address perturbed by the values of the red and green channels, and sampled from the map set on stage 0.

```
tex     t0        ; Define t0 to get a 3-tuple DuDvL.
texbeml t1, t0    ; Compute u = u1 + mat00*r0.r + mat01*r0.g, and
                  ;         v = v1 + mat10*r0.r + mat11*r0.g.
                  ; Then apply luminance correction.
mov r0, t1        ; Copy final result to output color.
```

The texture is set at the second stage, the texture set on the first stage is ignored.

This operation is a macro of two instructions (consuming 2 stages/slots): u' = u + du∗M00 + dv∗M01 and v' = v + du∗M10 + dv∗M11 where M00, M01, M10, and M11 are D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT01, D3DTSS_BUMPENVMAT10, and D3DTSS_BUMPENVMAT11.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texcoord

Converts the iterated texture coordinate assigned to this stage to a color for use in blending.

```
texcoord   tDest
```

Registers

tDest
Destination register, holding the result of the operation.

Remarks

This instruction allows the use of the texture coordinate as an additional Gouraud iterated (guaranteed perspective correct) color.

```
texcoord  t0    ; Declare register t0 as a color from its texture coordinates.
```

Because any quantity can be mapped by the vertex shader into a texture coordinate, the **texcoord** instruction can be used to stream arbitrary data such as position, normal, and light source direction to the pixel shader.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texkill

Masks out the pixel, if any texture coordinates are less than 0.

```
texkill   tDest
```

Registers

tDest
Destination register, holding the result of the operation.

This instruction can be used to implement arbitrary clip planes in the rasterizer.

When using vertex shaders, the application is responsible for applying the perspective transform. This can cause problems for the arbitrary clipping planes because if it contains anisomorphic scale factors, the clip planes would need to be transformed as well. Therefore, it is best to provide an unprojected vertex position for use in the arbitrary clipper, which is the texture coordinate set identified by the **texkill** operator.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texm3x2pad

Used in combination with other texture address operations to perform 2×3 matrix multiplies.

```
texm3x2pad   tDest, tSrc0
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Temporary source register, specifying the input argument.

Remarks

This instruction is used to represent stages where only the texture coordinate is used. These corresponding stages have not textures bound, and no sampling will occur. The input argument, t0, should still be specified.

The following example shows how the **texm3x2pad** instruction might be used.

```
tex t0            ; Define t0 as a standard 3-vector.
texm3x2pad t1, t0  ; Perform fist row of matrix multiply.
texm3x2tex t2, t0  ; Perform second row of matrix multiply to get a
                   ; 2-vector with which to sample texture 2.
```

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texm3x2tex

Performs a 3×2 matrix multiply on the input color vector.

```
texm3x2tex   tDest, tSrc0
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Temporary source register, specifying the input argument.

Remarks

The **texm3x2tex** instruction takes the specified input color and calculates the dot product of the input color and that stage's texture coordinates (u, v, and w) to produce a u coordinate. Then, it takes that same input color and calculates the dot product of the input color and the texture coordinates of this stage to compute the v coordinate. Lastly, this stage's texture is sampled at (u, v) to produce the final color. No texture is sampled or needs to be set at the preceding stage, and any operator assigned to it is overridden by this one.

The following example shows how the **texm3x2tex** instruction might be used.

```
tex t0          ; Define t0 as a standard 3-vector.
texm3x2pad t1, t0  ; Perform first row of matrix multiply.
texm3x2tex t2, t0  ; Perform second row of matrix multiply to get a
                ; 2-vector with which to sample texture 2.
mov r0, t2
```

A bump (normal) map should be set on stage 0, and a standard 2-D texture should be set on stage 2. Any texture set at stage 1 is ignored. For example, in the following shader, a base texture is lit by modulating with a diffuse bump map and adding an exponential specular. Stage 0 sets the base texture, stage 1 sets the normal map, and stage 2 sets the normal map again.

```
ps.1.0              ; DirectX8 Version
tex t0              ; sample normal map
texm3x2pad t1, t0_bx2  ; dot with light vector
texm3x2tex t2, t0_bx2  ; dot with halfway vector and sample
mov r0, t2
```

Texture coordinate set 0 positions the bump map. Texture coordinate sets 1 and 2 are the rows of the matrix.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texm3x3pad

Represents operations that are consumed by rows of the 3×3 matrix multiply that have no textures bound.

```
texm3x3pad   tDest, tSrc0
```

Registers

tDest

Destination register, holding the result of the operation.

tSrc0

Temporary source register, specifying the input argument.

Remarks

This instruction is a part of a 3×3 matrix multiply operation performed in a pixel shader's texture declaration stage. Texture coordinates corresponding to the declared texture are used as a row of the matrix. No texture should be bound at this stage.

The input argument to this declaration, t0, should be the input argument to the matrix multiply, **tSrc0**, as shown in the following example pixel shader.

```
tex t0          ; Define t0 as a standard 3-vector.
texm3x3pad t1, t0  ; Perform first row of matrix multiply.
texm3x3pad t2, t1  ; Perform second row of matrix multiply.
texm3x3tex t3, t2  ; Perform third row of matrix multiply to get a
                   ; 3-vector with which to sample texture 2.
mov r0, t3
```

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texm3x3spec

Performs specular reflection and environment mapping.

```
texm3x3spec   tDest, tSrc0, tSrc1
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

tSrc1
Source register, specifying the input argument.

Remarks

This operation takes the resulting post-transformed vector, and uses it as the normal to reflect an eye-ray vector. This eye-ray vector is always the constant color c0. Then, the instruction uses the result as an address to look up in a cube texture set at that stage. A specular environment map would be preloaded.

```
tex t0              ; Define t0 as a standard 3-vector.
texm3x3pad t1, t0      ; Perform first row of matrix multiply.
texm3x3pad t2, t0      ; Perform second row of matrix multiply.
texm3x3spec t3, t0, c0  ; Perform third row of matrix multiply, do
```

```
                ; reflection calculation, and sample texture 3.
mov r0, t3      ; Copy final result to output color.
```

A bump (normal) map should be set on stage 0, and cube texture should be set on stage 3. Any textures set at stages 1 or 2 are ignored.

Texture coordinate set 0 positions the bump map. Texture coordinate sets 1, 2, and 3 are the rows of the 3×3 matrix.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

See Also

texm3x3vspec

texm3x3tex

Returns the result of the final 3×3 matrix multiply.

```
texm3x3tex   tDest, tSrc0
```

tDest
Destination register, holding the result of the operation.

tSrc0
Temporary source register, specifying the input argument.

Remarks

The **texm3x3tex** instruction is used as the final of three instructions representing a 3×3 matrix multiply operation performed in a pixel shader's texture declaration. The result of this instruction will contain the resulting transformed vector. The input argument to this declaration should be the input to the matrix multiply. Texture coordinates assigned to this stage are used as a row of the matrix.

The 3×3 matrix is comprised of the texture coordinates of this and the two preceding stages. The resulting three-component-vector is then used to index into the texture assigned at this stage, (which can be either a cube texture or a volume texture) to produce the final color. No texture colors are sampled from the preceding two stages, and any operator assigned to them is overridden by this one.

```
tex t0          ; Define t0 as a standard 3-vector.
texm3x3pad t1, t0  ; Perform first row of matrix multiply.
texm3x3pad t2, t0  ; Perform second row of matrix multiply.
texm3x3tex t3, t0  ; Perform third row of matrix multiply.

dp3 r0, t0, v0     ; Compute lighting intensity for diffuse
                ; using v0.
```

A bump (normal) map should be set on stage 0, and cube texture should be set on stage 3. Any textures set at stages 1 or 2 are ignored.

Texture coordinate set 0 positions the bump map. Texture coordinate sets 1, 2, and 3 are the rows of the 3×3 matrix.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texm3x3vspec

Performs specular reflection and environment mapping where the eye-vector is not constant.

```
texm3x3vspec   tDest, tSrc0
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Source register, specifying the input argument.

Remarks

This instruction is used for texture addressing. It works just as **texm3x3spec**, except that the eye-ray is taken from the w-components of the three sets of texture coordinates used as rows of the matrix.

```
tex t0              ; Define t0 as a standard 3-vector.
texm3x3pad t1, t0   ; Perform first row of matrix multiply.
texm3x3pad t2, t0   ; Perform second row of matrix multiply.
texm3x3vspec t3, t0 ; Perform third row of matrix multiply, sample
                    ; texture 2, do reflection calculation
                    ; using eye ray, and sample texture 3
                    ; specular map.
mov r0.rgb, t3      ; Put specular color into the result.
mov r0.a,   t3      ; Put diffuse color into the alpha channel.
```

A bump (normal) map should be set on stage 0, and cube texture should be set on stage 3. Any textures set at stages 1 or 2 are ignored.

Texture coordinate set 0 positions the bump map. Texture coordinate sets 1, 2, and 3 are the rows of the 3×3 matrix.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

See Also

texm3x3spec

texreg2ar

Samples this stage's texture at the 2-D coordinates specified by the alpha and red components of the specified input color vector.

```
texreg2ar tDest, tSrc0
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Temporary source register, specifying the input argument.

Remarks

This instruction is useful for color space re-mapping operations.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

texreg2gb

Samples this stage's texture at the 2-D coordinates specified by the green and blue components of the specified input color vector.

```
texreg2gb tDest, tSrc0
```

Registers

tDest
Destination register, holding the result of the operation.

tSrc0
Temporary source register, specifying the input argument.

Remarks

This instruction is useful for color space re-mapping operations.

Note that this instruction has an unnamed input which is the assigned texture that corresponds to this texture stage.

Modifiers

The Direct3DX pixel shader assembler supports the following types of modifiers.

- Argument Modifiers
- Instruction Modifiers

Argument Modifiers

The Direct3DX pixel shader assembler can take instruction arguments (operands) with modifier flags that affect the input argument before it is processed in the instruction. There are two types of input argument modifiers: color and alpha blending input argument modifiers and texture addressing input argument modifiers.

The input argument modifiers are defined by the following table, where r_n is an input argument (register).

Instruction	Syntax	Description
Alpha Replicate	r_n.a	Replicates the alpha channel to all colors.
Invert	1-r_n	Complements y = 1.0 - x. Unsigned x is required.
Negate	-r_n	Negates the value y = -x. Signed x is required.
Bias	r_n_bias	Shifts value down by ½, y = (x-0.5).
Signed Scaling	r_n_bx2	Shifts value down and scales data by 2, y = 2∗(x-0.5).

For example, the **lrp** instruction can be implemented using input argument modifiers.

```
; Using diffuse alpha to blend between r0 and t0
lrp r0, v0.a, r0, t0
; can be written (using more clocks) as:
sub r0, r0, t0
mad r0, t0, v0.a, r0
```

Unlike vertex shaders, component swizzles are not supported by the pixel shader. The x, y, and z components are treated together as an RGB color component, and the w component is treated as alpha which can be routed separately.

Alpha Replicate

***r*_n.a**

Replicates the alpha channel of the affected register to all of its color channels before processing the instruction.

The following example shows how to use this modifier.

```
mul r0, r0, r1.a    ; Modulate by the alpha channel (grayscale),
                    ; varies only the intensity of color.
```

Note that this operator can be used in conjunction with the Invert or Negate operators. Its functionality is analogous to the D3DTA_ALPHAREPLICATE flag in the Microsoft® DirectX® 6.0 and 7.0 multitexture syntax.

This modifier is for use with the color and alpha blending instructions.

Invert

1-*r*$_n$

Compliments the colors in each channel of the specified register.

The following example shows how to use this modifier.

```
mul r0, r0, 1-r1    ; Multiply by (1.0 – r1).
```

This modifier only produces defined results when the input data is unsigned, meaning that it is in the range of 0 to 1. This operation is performed after any other modifiers present on the same argument.

Note that this modifier only works with unsigned data. The use of this instruction is inconsistent with the Bias, Signed Scaling, and Negate modifiers, so it cannot appear on the same register with any of these modifiers.

This modifier is for use with the color and alpha blending instructions.

Negate

-*r*$_n$

Performs a signed inverse (y = -x), or negates the value before it is used in the instruction.

The following example shows how to use this modifier.

```
mul r0, r0, -v1    ; Multiply by –specular color.
```

This operation is performed after any other modifiers present on the same argument.

Note that Invert (1-***r***$_n$)and Negate (-***r***$_n$) are mutually exclusive and cannot be applied to the same register.

This modifier is for use with the color and alpha blending instructions.

Bias

r$_n$**_bias**

Shift each channel down by ½. So, it performs y = (x-0.5) before the register is operated on.

The following example shows how to use this modifier.

```
add r0, r0, t0_bias    ; Shift down by 0.5.
```

This example shows how to use **add** to perform the same operation as D3DTOP_ADDSIGNED in Microsoft® DirectX® 6.0 and 7.0 multitexture syntax.

This modifier has the effect of modifying data that was in the range 0 to 1 to be in the range -0.5 to 0.5. This is commonly used for applying detail textures. It can also enable the input data to be processed as a signed quantity while reserving dynamic range before overflow clamping that can occur on implementations limited to the range -1 to1.

Note that this modifier is mutually exclusive with Invert, so it cannot be applied to the same register.

This modifier is for use with the color and alpha blending instructions.

Signed Scaling

***r*$_n$_bx2**

When used with a color and alpha blending instruction, this modifier subtracts 0.5 from each channel and scales the result by 2.0 before the instruction is executed. So it performs y = (x-0.5)✶2.

The following example shows how to use this modifier.

```
dp3_sat r0, t1_bx2, v0_bx2    ; Per-pixel lighting/bump mapping which assumes bump normals
                              ; assumes bump normals in t1 and light direction in diffuse
                              ; color v0.
```

This modifier has the effect of remapping data that was in the range 0 to 1 to be in the signed range -1 to 1. This enables subsequent processing of the data to use the full signed dynamic range of the implementation.

This modifier is commonly used on inputs to the dot product instruction (**dp3**).

Note that this modifier is mutually exclusive with Invert, so it cannot be applied to the same register.

The input arguments to texture register declarations can support the **_bx2** modifier to indicate additional processing to be performed during the execution of the declaration/instruction for the **dp3**, **texm3x2tex**, and **texm3x3tex** operations. The other texture addressing operators do not support the **_bx2** input argument modifiers.

To indicate that the input register argument should be biased and scaled before use, append **_bx2** to the input register argument.

```
tex t0          ; Read a texture color.
dp3 t1, t0_bx2  ; Calculate a dotproduct with this stage's texture coordinates.
```

This performs the operation y = 2(x - 0.5) on the input register before using it in the specified operator. This operation is commonly used to translate data that is intended to contain signed values in the range -1 to 1, but does not do this because the source it comes from cannot support it. Such data sources include textures (t*n*), iterated colors (v*n*), texture coordinates, or constant factor registers (c*n*).

This modifier is for use with both the color and alpha blending instructions and the texture addressing instructions.

Instruction Modifiers

The Direct3DX pixel shader assembler support shift/scale modifier flags and a saturation modifier flag that affects the generated output result. Instruction modifiers integrate with the instruction processing in the following order.

1. Perform inversion.
2. Apply biasing.
3. Apply scaling to the result.
4. Perform clamping.

The shift/scale modifiers can vary the magnitude of the result within a certain range.

Name	Modifier
_x2	Multiply result by 2
_x4	Multiply result by 4
_d2	Divide result by 2

In addition, a saturation modifier (**_sat**) is provided for both color and alpha blending instructions and texture addressing instructions.

For color and alpha blending instruction, the saturation modifier clamps the result of this instruction into the range 0.0 to 1.0 for each component. The following example shows how to use this instruction modifier.

```
dp3_sat r0, t0_bx2, v0_bx2    ; t0 is bump, v0 is light direction
```

This operation occurs after the scaling instruction modifier. This is most often used to clamp dot product results. However, it also enables consistent emulation of multipass methods where the frame buffer is always in the range 0 to 1, and of Microsoft® DirectX 6.0 and 7.0 multitexture syntax, in which saturation is defined to occur at every stage.

For texture addressing instructions, the saturation modifier can be appended to the operation to indicate that the result of the operation should be clamped into the color range 0 .0 to 1.0 upon completion of the operation. This is commonly required for use in lighting operations. The following example shows how to use this instruction modifier.

```
ps.1.0                  ; DirectX8 Version
tex t0                  ; Read a texture color.
dp3_sat t1, t0_bx2, v0_bx2   ; Calculate a dotproduct with the texture coordinates and clamp.
```

The following **add** instruction loads the destination register (d) with the sum of the two colors in the source operands (s0 and s1), and multiplies the result by 2.

```
add_x2   d, s0, s1
```

The following **add** instruction loads the destination register (d) with the sum of the two colors in the source operands (s0 and s1), and clamps the result into the range 0.0 to 1.0 for each component.

```
add_sat   d, s0, s1
```

The following **add** instruction loads the destination register (d) with the sum of the two colors in the source operands (s0 and s1), multiplies the result by 2, and clamps the result into the range 0.0 to 1.0 for each component.

```
add_2x_sat   d, s0, s1
```

Output Write Masks

The output masks supported by the Direct3DX pixel shader compiler are used to indicate whether this result should update either the color or the alpha component of the result, and therefore whether to use the color or alpha pipelines for its processing. These output masks are analogous to the separate color and alpha operations in Microsoft® DirectX 6.0 and 7.0 multitexture blending.

These flags are not as general as those supported by vertex shaders because the rgb components are always grouped into a single flag, resulting in only two different masks, .a and .rgb.

The following output write masks are supported.

- Color/Vector Write Mask
- Alpha/Scalar Write Mask
- Full Write Mask

Note that none of the output write masks are supported for the texture addressing instruction. These operations always update all four components of the destination register, you cannot separate r, g, and b individually.

The pixel shader does not support arbitrary swizzles, only color and alpha operations.

Color/Vector Write Mask

.rgb

When specified on the output register, this flag indicates that the operation will update only the three color channels of the destination register, and therefore should be performed in the color (vector) pipeline for the instruction.

The following example shows how to use this output write mask.

```
mul r0.rgb,  t0, v0
add r0.a,    t1, v1
```

Alpha/Scalar Write Mask

.a

When specified on the destination register, this flag indicates that the operation will update only the alpha channel of the destination register, and therefore should be performed in the alpha (scalar) pipeline for this instruction.

The following example shows how to use this output write mask.

```
mul r0.rgb,  t0, v0
add r0.a,    t1, v1
```

Full Write Mask

.rgba

Requests that the same instruction be applied in both the color and alpha channels. This is the default, so no output mask needs to be specified to achieve the same result.

For example, the following instruction

```
mul r0, t0, v0
```

is equivalent to

```
mul r0.rgba, t0, v0
```

which is also equivalent to the following instruction.

```
  mul r0.rgb, t0, v0
+ mul r0.a,   t0, v0
```

Note that even this last syntax takes only one clock due to pairing between the scalar and vector pipelines. For more information, see Pixel Shader Instruction Pairing.

Effect File Format

[Visual Basic]

This topic pertains only to applications written in C++.

[C++]
This section contains reference information for the effect file format. An effect file consists of a set of parameter declarations, followed by descriptions of various techniques.

- Parameter Types
- Constant Value Syntax
- Vertex Shader Declaration Syntax
- List of Valid States
- Sample of an Effect File

Technique descriptions have the following syntax. Remember that *id* must be a valid FourCC code. For information on FourCC codes, see Four-Character Codes (FOURCC).

- Variable declarations go before the body of the technique.
- The body of the technique contains one or more pass descriptions.
- A pass contains one or more state assignments.
- States can be assigned to either a constant value or to a parameter.

```
{type} {id};
{type} {id} = {const};

TECHNIQUE {id}
{
   PASS {id}
   {
      {state}      = {const};
      {state}[{n}] = {const};
      {state}      = <{id}>;
      {state}[{n}] = <{id}>;
   }
}
```

Parameter Types

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
Parameter declarations have the following syntax, where *type* is a valid type, and *id* is a valid Four-Character Codes (FOURCC). If a *const* value is provided as an initial value for the parameter, its type must match *type*.

```
{type} {id};
{type} {id} = {const};
```

The following table lists all of the valid parameter types that are used for effect file parameter declarations along with a sample.

Type	Sample
DWORD	DWORD minVertices;
FLOAT	FLOAT fRotationAdvances;
VECTOR	VECTOR vecPoint;
MATRIX	MATRIX matIndentity;
TEXTURE	TEXTURE tex1;
VERTEXSHADER	VERTEXSHADER v1;
PIXELSHADER	PIXELSHADER p1;

Constant Value Syntax

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The following table list shows the proper syntax for constant values and a sample.

Type		Syntax
DWORD	#	54573153
	0x#	0xff12a1fa
FLOAT	#f	5f
	#.f	5.f
	.#f	.4f
	#.#f	4.5f
VECTOR	(float)	(4.5f)
	(float, float)	(4.5f, 1.0f)
	(float, float, float)	(4.5f, 1.0f, 2.0f)
	(float, float, float, float)	(4.5f, 1.0f, 2.0f, 3.4f)
MATRIX	[float, float, float, float,	[1.0f, 0.0f, 0.0f, 0.0f,

	float, float, float, float, float, float, float, float, float, float, float, float]	0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f]

VERTEXSHADER	DECL { ... }	decl
		{
		stream 0;
		float v0[3];
		float v3[3];
		ubyte v5[4];
		float v7[2];
		}

	DECL { ... }, ASM { ... }	decl
		{
		stream 0;
		float v0[3];
		float v3[3];
		ubyte v5[4];
		float v7[2];
		}
		asm
		{
		vs.1.0
		mov oPos, v0
		}

PIXELSHADER	ASM { ... }	asm
		{
		vs.1.0
		mov oPos, v0
		}

For information on the layout of the DECL syntax, see Vertex Shader Declaration Syntax.

For information on the layout of the ASM syntax, see Pixel Shader Assembler Reference.

Vertex Shader Declaration Syntax

[Visual Basic]
This topic pertains only to applications written in C++.

```
[C++]
  DECL
  {
     STREAM n;
     SKIP n;
     FVF a|b|c;

     type v#;
     type v#[n];
  }
```

The following table describes the type of values used for the above syntax.

Syntax	Description
a|b|c	**DWORD** expression which can consist of **DWORD**s and flexib vector format flags. To use a FVF, just strip off the D3DFVF_ prefix.
Type	One of the following types: UBYTE, SHORT, FLOAT, D3DCOLOR.
v#	A valid vertex shader input register. See the table below for the of FVF flags and their corresponding vertex shader input registe
n	A number between 1 and 4

The following table lists the FVF codes and the corresponding vertex shader input registers.

FVF	Name	Register Number
D3DFVF_XYZ	D3DVSDE_POSITION	0
D3DFVF_XYZRHW	D3DVSDE_BLENDWEIGHT	1
D3DFVF_XYZB1 through D3DFVF_XYZB5	D3DVSDE_BLENDINDICES	2
D3DFVF_NORMAL	D3DVSDE_NORMAL	3
D3DFVF_PSIZE	D3DVSDE_PSIZE	4
D3DFVF_DIFFUSE	D3DVSDE_DIFFUSE	5
D3DFVF_SPECULAR	D3DVSDE_SPECULAR	6
D3DFVF_TEX0	D3DVSDE_TEXCOORD0	7
D3DFVF_TEX1	D3DVSDE_TEXCOORD1	8
D3DFVF_TEX2	D3DVSDE_TEXCOORD2	9
D3DFVF_TEX3	D3DVSDE_TEXCOORD3	10
D3DFVF_TEX4	D3DVSDE_TEXCOORD4	11
D3DFVF_TEX5	D3DVSDE_TEXCOORD5	12

D3DFVF_TEX6	D3DVSDE_TEXCOORD6	13
D3DFVF_TEX7	D3DVSDE_TEXCOORD7	14
	D3DVSDE_POSITION2	15
	D3DVSDE_NORMAL2	16

List of Valid States

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The following topic covers the entire list of valid states that can be used when creating an effect file.

When setting states of type **DWORD**, you can use enumerated values. For example, SrcBlend is typically set using D3DBLEND_xxx enumerated types. Therefore, you can use any of these enumeration types in your assignment. Just strip off the D3DBLEND_ prefix as shown in the code fragment below.

```
SrcBlend = One;
```

You can also string these together in an "OR" expression.

```
TextureTransformFlags[0] = Count3 | Projected;
```

The following table lists the valid render states used in an effect file.

Type	Render State	Valid Values
DWORD	ZEnable	D3DZB
DWORD	FillMode	D3DFILL
DWORD	ShadeMode	D3DSHADE
DWORD	LinePattern	
DWORD	ZWriteEnable	TRUE/FALSE
DWORD	AlphaTestEnable	TRUE/FALSE
DWORD	LastPixel	TRUE/FALSE
DWORD	SrcBlend	D3DBLEND
DWORD	DestBlend	D3DBLEND
DWORD	CullMode	D3DCULL
DWORD	ZFunc	D3DCMP

DWORD	AlphaRef	
DWORD	AlphaFunc	D3DCMP
DWORD	DitherEnable	TRUE/FALSE
DWORD	AlphaBlendEnable	TRUE/FALSE
DWORD	FogEnable	TRUE/FALSE
DWORD	SpecularEnable	TRUE/FALSE
DWORD	ZVisible	TRUE/FALSE
DWORD	FogColor	
DWORD	FogTableMode	D3DFOG
FLOAT	FogStart	
FLOAT	FogEnd	
FLOAT	FogDensity	
DWORD	EdgeAntialias	TRUE/FALSE
DWORD	Zbias	
DWORD	RangeFogEnable	
DWORD	StencilEnable	TRUE/FALSE
DWORD	StencilFail	D3DSTENCILOP
DWORD	StencilZFail	D3DSTENCILOP
DWORD	StencilPass	D3DSTENCILOP
DWORD	StencilFunc	D3DCMP
DWORD	StencilRef	
DWORD	StencilMask	
DWORD	StencilWriteMask	
DWORD	TextureFactor	
DWORD	Wrap0	D3DWRAP
DWORD	Wrap1	D3DWRAP
DWORD	Wrap2	D3DWRAP
DWORD	Wrap3	D3DWRAP
DWORD	Wrap4	D3DWRAP
DWORD	Wrap5	D3DWRAP
DWORD	Wrap6	D3DWRAP
DWORD	Wrap7	D3DWRAP
DWORD	Clipping	TRUE/FALSE
DWORD	Lighting	TRUE/FALSE
DWORD	Ambient	
DWORD	FogVertexMode	
DWORD	ColorVertex	TRUE/FALSE
DWORD	LocalViewer	TRUE/FALSE

DWORD	NormalizeNormals	TRUE/FALSE
DWORD	DiffuseMaterialSource	D3DMCS
DWORD	SpecularMaterialSource	D3DMCS
DWORD	AmbientMaterialSource	D3DMCS
DWORD	EmissiveMaterialSource	D3DMCS
DWORD	VertexBlend	D3DVBF
DWORD	ClipPlaneEnable	TRUE/FALSE
DWORD	SoftwareVertexProcessing	TRUE/FALSE
FLOAT	PointSize	
FLOAT	PointSize_Min	
FLOAT	PointSize_Max	
DWORD	PointSpriteEnable	TRUE/FALSE
DWORD	PointScaleEnable	TRUE/FALSE
FLOAT	PointScale_A	
FLOAT	PointScale_B	
FLOAT	PointScale_C	
DWORD	MultiSampleAntialias	TRUE/FALSE
DWORD	MultiSampleMask	
DWORD	PatchSegments	
DWORD	IndexedVertexBlendEnable	TRUE/FALSE
DWORD	ColorWriteEnable	TRUE/FALSE
FLOAT	TweenFactor	
DWORD	BlendOp	D3DBLENDOP

The following table lists the valid texture stage states used in an effect file.

Type	**Texture Stage State**	**Valid Values**
DWORD	ColorOp[8]	D3DTOP
DWORD	ColorArg0[8]	D3DTA
DWORD	ColorArg1[8]	D3DTA
DWORD	ColorArg2[8]	D3DTA
DWORD	AlphaOp[8]	D3DTOP
DWORD	AlphaArg0[8]	D3DTA
DWORD	AlphaArg1[8]	D3DTA
DWORD	AlphaArg2[8]	D3DTA
DWORD	ResultArg[8]	D3DTA
DWORD	BumpEnvMat00[8]	
DWORD	BumpEnvMat01[8]	

DWORD	BumpEnvMat10[8]	
DWORD	BumpEnvMat11[8]	
DWORD	TexCoordIndex[8]	D3DTSS_TCI
DWORD	AddressU[8]	D3DTADDRESS
DWORD	AddressV[8]	D3DTADDRESS
DWORD	AddressW[8]	D3DTADDRESS
DWORD	BorderColor[8]	
DWORD	MagFilter[8]	D3DTEXF
DWORD	MinFilter[8]	D3DTEXF
DWORD	MipFilter[8]	D3DTEXF
DWORD	MipMapLodBias[8]	
DWORD	MaxMipLevel[8]	
DWORD	MaxAnsiotropy[8]	
FLOAT	BumpEnvLScale[8]	
FLOAT	BumpEnvLOffset[8]	
DWORD	TextureTransformFlags[8]	D3DTTFF

The following table lists the valid light states used in an effect file. If you set light states in your effect, you should set all states that you need to completely describe the light. States that you fail to declare will be set to some default value because there is no way for D3DX to set light states individually.

Type	**Light State**	**Valid Values**
DWORD	LightType[n]	D3DLT
VECTOR	LightDiffuse[n]	
VECTOR	LightSpecular[n]	
VECTOR	LightAmbient[n]	
VECTOR	LightPosition[n]	
VECTOR	LightDirection[n]	
FLOAT	LightRange[n]	
FLOAT	LightFalloff[n]	
FLOAT	LightAttenuation0[n]	
FLOAT	LightAttenuation1[n]	
FLOAT	LightAttenuation2[n]	
FLOAT	LightTheta[n]	
FLOAT	LightPhi[n]	
DWORD	LightEnable[n]	TRUE/FALSE

The following table lists the valid material states used in an effect file. If you set material states in your effect, you should set all states that you need to completely

describe the material. States that you fail to declare will be set to some default value because there is no way for D3DX to set material states individually.

Type	Material State
VECTOR	MaterialDiffuse
VECTOR	MaterialAmbient
VECTOR	MaterialSpecular
VECTOR	MaterialEmissive
FLOAT	MaterialPower

The following table lists other valid states used in an effect file.

Type	State
VERTEXSHADER	VertexShader
PIXELSHADER	PixelShader
CONSTANT	VertexShaderConstant[n]
CONSTANT	PixelShaderConstant[n]
TEXTURE	Texture[8]
MATRIX	ProjectionTransform
MATRIX	ViewTransform
MATRIX	WorldTransform[256]
MATRIX	TextureTransform[8]

Sample of an Effect File

[Visual Basic]
This topic pertains only to applications written in C++.

[C++]
The following shows the basic syntax and layout of a typical effect file.

```
//
// Sample Effect
// This effect adds two textures, using single pass or multipass technique.
//

texture tex0;
texture tex1;
```

```
// Single pass

technique t0
{
   pass p0
   {
      Texture[0] = <tex0>;
      Texture[1] = <tex1>;

      ColorOp[0] = SelectArg1;
      ColorArg1[0] = Texture;

      ColorOp[1] = Add;
      ColorArg1[0] = Texture;
      ColorArg2[0] = Current;

      ColorOp[2] = Disable;
   }
}

// Multipass

technique t1
{
   pass p0
   {
      Texture[0] = <tex0>;

      ColorOp[0] = SelectArg1;
      ColorArg1[0] = Texture;
      ColorOp[1] = Disable;
   }

   pass p1
   {
      AlphaBlendEnable = True;
      SrcBlend = One;
      DestBlend = One;

      Texture[0] = <tex1>;

      ColorOp[0] = SelectArg1;
      ColorArg1[0] = Texture;
      ColorOp[1] = Disable;
```

```
  }
}
```

Direct3D Visual Basic Reference

This section contains reference information for the API elements provided by Microsoft® Direct3D®. Reference material is divided into the following categories.

- Classes
- Functions
- Vertex Shader Declarator Functions
- Types
- Enumerations
- Texture Argument Flags
- Flexible Vertex Format Flags
- Four-Character Codes (FOURCC)
- Error Codes

Classes

This section contains reference information for the classes provided by Microsoft® Direct3D®. The following classes are used with Direct3D.

- **Direct3D8**
- **Direct3DBaseTexture8**
- **Direct3DCubeTexture8**
- **Direct3DDevice8**
- **Direct3DIndexBuffer8**
- **Direct3DResource8**
- **Direct3DSurface8**
- **Direct3DSwapChain8**
- **Direct3DTexture8**
- **Direct3DVertexBuffer8**
- **Direct3DVolume8**
- **Direct3DVolumeTexture8**

Direct3D8

#Applications use the methods of the **Direct3D8** class to create Microsoft® Direct3D® objects and set up the environment. This class includes methods for enumerating and retrieving capabilities of the device.

The **Direct3D8** class is obtained by calling the **DirectX8.Direct3DCreate** method.

The methods of the **Direct3D8** class can be organized into the following groups.

Creation	**CreateDevice**
Enumeration	**EnumAdapterModes**
Information	**GetAdapterCount**
	GetAdapterDisplayMode
	GetAdapterIdentifier
	GetAdapterModeCount
	GetAdapterMonitor
	GetDeviceCaps
Registration	**RegisterSoftwareDevice**
Verification	**CheckDepthStencilMatch**
	CheckDeviceFormat
	CheckDeviceMultiSampleType
	CheckDeviceType

See Also

DirectX8.Direct3DCreate

Direct3D8.CheckDepthStencilMatch

#Determines whether a surface format is available as a specified resource type and can be used as a texture, depth-stencil buffer, or render target, or any combination of the three, on a device representing this adapter.

```
object.CheckDepthStencilMatch( _
    Adapter As Long, _
    DeviceType As  CONST_D3DDEVTYPE, _
    AdapterFormat As CONST_D3DFORMAT, _
```

IDH_Direct3D8_graphicsvb
IDH_Direct3D8.CheckDepthStencilMatch_graphicsvb

RenderTargetFormat **As CONST_D3DFORMAT**, _
DepthStencilFormat **As CONST_D3DFORMAT) As Long**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter
Ordinal number denoting the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter.

DeviceType
Member of the **CONST_D3DDEVTYPE** enumeration identifying the device type.

AdapterFormat
Member of the **CONST_D3DFORMAT** enumeration, identifying the format of the display mode into which the adapter will be placed.

RenderTargetFormat
Member of the **CONST_D3DFORMAT** enumeration, identifying the format of the render target surface to be tested.

DepthStencilFormat
Member of the **CONST_D3DFORMAT** enumeration, identifying the format of the depth-stencil surface to be tested.

Return Values

If the depth-stencil format is compatible with the render target format in the display mode, this method returns D3D_OK.

D3DERR_INVALIDCALL can be returned if one or more of the parameters is invalid. If a depth-stencil format is not compatible with the render target in the display mode, then this method returns D3DERR_NOTAVAILABLE.

Error Codes

Err.Number is not set for this method.

Remarks

This method is provided to allow applications to work with hardware requiring that certain depth formats can only work with certain render target formats. The following code fragment shows how you could use CheckDeviceFormat to validate a depth stencil format.

```
Function IsDepthFormatOK( DepthFormat As CONST_D3DFORMAT, AdapterFormat As
CONST_D3DFORMAT, _
```

```
                BackBufferFormat As CONST_D3DFORMAT) as Boolean

    Dim check As Long

    check = d3d.CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
AdapterFormat, _
                        D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE, DepthFormat)
    If (check < 0) Then Exit Function   'Returns False

    check = d3d.CheckDepthStencilFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
AdapterFormat, _
                RenderTargetFormat, DepthFormat)
    If (check < 0) Then
      IsDepthStencilFormatOK = False
    Else
      IsDepthStencilFormatOK = True
    End If

End function
```

The preceding call returns False if *DepthFormat* cannot be used in conjunction with *AdapterFormat* and *BackBufferFormat*.

Direct3D8.CheckDeviceFormat

#Determines whether a surface format is available as a specified resource type and can be used as a texture, depth-stencil buffer, or render target, or any combination of the three, on a device representing this adapter.

object.**CheckDeviceFormat(_**
Adapter **As Long**, **_**
DeviceType **As CONST_D3DDEVTYPE**, **_**
AdapterFormat **As CONST_D3DFORMAT**, **_**
Usage **As Long**, **_**
RType **As CONST_D3DRESOURCETYPE**, **_**
CheckFormat **As CONST_D3DFORMAT) As Long**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter
Ordinal number denoting the display adapter to query.
D3DADAPTER_DEFAULT is always the primary display adapter. This method

IDH_Direct3D8.CheckDeviceFormat_graphicsvb

returns D3DERR_INVALIDCALL when this value equals or exceeds the number of display adapters in the system.

DeviceType
Member of the **CONST_D3DDEVTYPE** enumeration, identifying the device type.

AdapterFormat
Member of the **CONST_D3DFORMAT** enumeration, identifying the format of the display mode into which the adapter will be placed.

Usage
Requested usage for a surface of the *CheckFormat*. One or more of the following flags defined by the **CONST_D3DUSAGEFLAGS** enumeration can be specified.

D3DUSAGE_DEPTHSTENCIL
Set to indicate that the surface can be used as a depth-stencil surface.

D3DUSAGE_RENDERTARGET
Set to indicate that the surface can be used as a render target. In cases where the *RType* parameter value unambiguously implies a specific usage, the application can leave the *Usage* parameter as 0.

RType
A member of the **CONST_D3DRESOURCETYPE** enumeration, specifying the resource type requested for use with the queried format.

CheckFormat
Member of the **CONST_D3DFORMAT** enumeration. Indicates the format of the surfaces that may be used, as defined by *Usage*.

Return Values

If the format is compatible with the specified device for the requested usage, this method returns D3D_OK.

D3DERR_INVALIDCALL is returned if *Adapter* equals or exceeds the number of display adapters in the system, or if *DeviceType* is unsupported. This method returns D3DERR_NOTAVAILABLE if the format is not acceptable to the device for this usage.

Error Codes

Err.Number is not set for this method.

Remarks

A typical use of **CheckDeviceFormat** is to verify the existence of a particular depth-stencil surface format. See Selecting a Device for more detail on the enumeration process. Another typical use of **CheckDeviceFormat** would be to verify if textures existing in particular surface formats can be rendered, given the current display mode.

```
Function DoesDepthFormatExist(DepthFormat As CONST_D3DFORMAT, AdapterFormat As
CONST_D3DFORMAT) As Boolean

  Dim Check As Long
  Dim D3D As Direct3D8

  'we assume the D3D object has been created and initialized
  Check = D3D.CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
AdapterFormat, _
                      D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE, DepthFormat)
  If (Check >= 0) Then
     DoesDepthFormatExist= True
  Else
    DoesDepthFormatExist= False
  End If

End Function
```

The preceding function call returns False if *DepthFormat* does not exist on the system.

Another typical use of **CheckDeviceFormat** would be to verify if textures existing in particular surface formats can be rendered, given the current display mode. The following code fragment shows how you could use **CheckDeviceFormat** to verify that texture formats are compatible with specific back buffer formats.

```
Function IsTextureFormatOk(TextureFormat As D3DFORMAT ,AdapterFormat As
D3DFORMAT ) As Boolean
  Dim Check As Long
  Dim D3D As Direct3D8

  Check = D3D.CheckDeviceFormat( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
AdapterFormat, _
                      0, D3DRTYPE_TEXTURE, TextureFormat)

  If (Check >= 0) Then
    IsTextureFormatOk = True
  Else
    IsTextureFormatOk = False
  End If

End Function
```

The preceding call returns FALSE if *TextureFormat* cannot be used to render textures while the adapter surface format is *AdapterFormat*.

Direct3D8.CheckDeviceMultiSampleType

[#]Determines if a multisampling technique is available on this device.

object.**CheckDeviceMultiSampleType(** _
Adapter **As Long**, _
DeviceType **As CONST_D3DDEVTYPE**, _
Format **As CONST_D3DFORMAT**, _
Windowed **As Long**, _
MultiSampleType **As CONST_D3DMULTISAMPLE_TYPE) As Long**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter
Ordinal number denoting the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter. This method returns FALSE (0) when this value equals or exceeds the number of display adapters in the system.

DeviceType
Member of the **CONST_D3DDEVTYPE** enumeration, identifying the device type.

Format
Member of the **CONST_D3DFORMAT** enumeration that specifies the set of multisampling types requested for the surface. See Remarks.

Windowed
Specify True to inquire about windowed multisampling, and specify False to inquire about full-screen multisampling.

MultiSampleType
Member of the **CONST_D3DMULTISAMPLE_TYPE** enumeration, identifying the multisampling technique to test.

Return Values

If the device can perform the specified multisampling method, this method returns D3D_OK.

If the method fails, one of the following values can be returned.

D3DERR_INVALIDCALL

D3DERR_INVALIDDEVICE

D3DERR_NOTAVAILABLE

[#] IDH_Direct3D8.CheckDeviceMultiSampleType_graphicsvb

Remarks

This method is intended for use with both render target and depth-stencil surfaces because you need to create both of them multisampled if you want to use them together.

The following code fragment shows how you could use **CheckDeviceMultiSampleType** to test for devices that support a specific multisampling method.

```
Dim Device As Direct3DDevice8
Dim Caps As D3DCAPS8
Dim Caps2 As CONST_D3DDEVTYPE
Dim Mode As CONST_D3DFORMAT

If Device.CheckDeviceMultiSampleType(Caps.AdapterOrdinal, Caps2.DeviceType, _
                        Mode.Format, False, D3DMULTISAMPLE_3_SAMPLES) Then
   ' things to do if D3DMULTISAMPLE_3_SAMPLES works in full screen mode
   ' with the specified surface format
End If
```

Error Codes

Err.Number is not set for this method.

Direct3D8.CheckDeviceType

#Verifies whether or not a certain device type can be used on this adapter and expect hardware acceleration using the given formats.

object.**CheckDeviceType(** _
   Adapter **As Long**, _
   CheckType **As CONST_D3DDEVTYPE**, _
   DisplayFormat **As CONST_D3DFORMAT, _**
   BackBufferFormat **As CONST_D3DFORMAT, _**
   bWindowed **As Long) As Long**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter
Ordinal number denoting the display adapter to enumerate. D3DADAPTER_DEFAULT is always the primary display adapter. This method

IDH_Direct3D8.CheckDeviceType_graphicsvb

returns D3DERR_INVALIDCALL when this value equals or exceeds the number of display adapters in the system.

CheckType
Member of the **CONST_D3DDEVTYPE** enumeration, indicating the device type to check.

DisplayFormat
Member of the **CONST_D3DFORMAT** enumeration, indicating the format of the adapter display mode for which the device type is to be checked. For example, some devices operate only in 16-bits-per-pixel modes.

BackBufferFormat
Member of the **CONST_D3DFORMAT** enumeration, indicating the format of the adapter display mode for which the device type is to be checked. For example, some devices operate only in 16-bits-per-pixel modes.

bWindowed
Value indicating whether the device type will be used in full-screen or windowed mode. If set to True, the query is performed for windowed applications; otherwise, this value should be set False.

Return Values

If the device can be used on this adapter, D3D_OK is returned.

D3DERR_INVALIDCALL is returned if *Adapter* equals or exceeds the number of display adapters in the system. This method returns D3DERR_INVALIDDEVICE if *CheckType* specified a device that does not exist. D3DERR_NOTAVAILABLE is returned if either surface format is not supported, or if hardware acceleration is not available for the specified formats.

Error Codes

Err.Number is not set for this method.

Remarks

The most important device type that may not be present is D3DDEVTYPE_HAL. D3DDEVTYPE_HAL requires hardware acceleration. Applications should use **CheckDeviceType** to determine if the needed hardware and drivers are present on the system. The other device that may not be present is D3DDEVTYPE_SW. This device type represents a pluggable software device that was registered using **Direct3D8.RegisterSoftwareDevice**.

Applications should not specify a *DisplayFormat* that contains an alpha channel. This will result in a failed call. Note that an alpha channel may be present in the back buffer, but the two display formats must be identical in all other respects. For example, if *DisplayFormat* is D3DFMT_X1R5G5B5, valid values for *BackBufferFormat* include D3DFMT_X1R5G5B5 and D3DFMT_A1R5G5B5, but exclude D3DFMT_R5G6B5.

The following code fragment checks to see if the default adapter will handle RGB565 colors in windowed mode using hardware acceleration.

```
    Dim DisplayFormat As CONST_D3DFORMAT, BackBufferFormat As
CONST_D3DFORMAT
    Dim IsWindowed As Long, bModeOK As Boolean, Check As Long

        DisplayFormat = D3DFMT_R5G6B5
        BackBufferFormat = D3DFMT_R5G6B5
    IsWindowed = 1     'TRUE

    Check = d3d.CheckDeviceType(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
DisplayFormat, BackBufferFormat, IsWindowed)
    If (Check >= 0) Then
       bModeOK = True
    Else
       bModeOK = False
    End If
```

Direct3D8.CreateDevice

#Creates a device to represent the display adapter.

object.**CreateDevice(** _
Adapter **As Long**, _
DeviceType **As CONST_D3DDEVTYPE**, _
hFocusWindow **As Long**, _
BehaviorFlags **As Long**, _
PresentationParameters **As D3DPRESENT_PARAMETERS) As Direct3DDevice8**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter
Ordinal number that denotes the display adapter. D3DADAPTER_DEFAULT is always the primary display adapter.

DeviceType
Member of the **CONST_D3DDEVTYPE** enumeration. Denotes the desired device type. If the desired device type is not available, the method will fail.

hFocusWindow

IDH_Direct3D8.CreateDevice_graphicsvb

Window handle to which focus belongs for this Direct3Ddevice. The window specified must be a top-level window for full-screen.

BehaviorFlags

A combination of one or more flags defined by the **CONST_D3DCREATEFLAGS** enumeration that control global behaviors of the Microsoft® Direct3D® device.

PresentationParameters

A **D3DPRESENT_PARAMETERS** type, describing the presentation parameters for the device to be created. Calling **CreateDevice** can change the value of the **BackBufferCount** member of **D3DPRESENT_PARAMETERS**; the back buffer count is changed to reflect a corrected number of back buffers.

Return Values

This method returns a **Direct3DDevice8** object, representing the created device.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method returns a fully working device object, set to the required display mode (or windowed), and allocated with the appropriate back buffers. The application needs only to create and set a depth buffer, if desired, to begin rendering.

See Also

Direct3DDevice8.Reset, **D3DDEVICE_CREATION_PARAMETERS**

Direct3D8.EnumAdapterModes

#Enumerates the display modes of an adapter.

```
object.EnumAdapterModes( _
    Adapter As Long, _
    Mode As Long, _
```

IDH_Direct3D8.EnumAdapterModes_graphicsvb

DisplayMode **As D3DDISPLAYMODE)**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter
Ordinal number that denotes the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter.

Mode
Ordinal number that denotes the mode to enumerate. D3DCURRENT_DISPLAY_MODE is a special value denoting the current desktop display mode on this adapter and cannot be specified for this method.

DisplayMode
A **D3DDISPLAYMODE** type, to be filled with information describing this mode.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Some display drivers include refresh rate information that will greatly increase the number of modes. On older drivers, refresh rates of D3DADAPTER_DEFAULT are the only ones enumerated. Direct3D attempts to limit display modes and refresh rates based on the monitor capabilities. However, there is still some risk that an enumerated mode will not be supported by the system's monitor. You may want to implement a display mode testing facility similar to that used by the Microsoft® Windows® Display Control Panel utility.

Use **Direct3D8.GetAdapterModeCount** to determine the total number of modes supported by the system. Valid values for the *Mode* parameter range from 0 to the total number of modes minus 1.

See Also

Direct3D8.GetAdapterModeCount

Direct3D8.GetAdapterCount

#Returns the number of adapters on the system.

object.**GetAdapterCount() As Long**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Return Values

A **LONG** value that denotes the number of adapters on the system at the time this **Direct3D8** object was created.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3D8.GetAdapterIdentifier

Direct3D8.GetAdapterDisplayMode

#Retrieves the current display mode of the adapter.

object.**GetAdapterDisplayMode(_**
Adapter **As Long**, **_**
Mode **As D3DDISPLAYMODE)**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter

IDH_Direct3D8.GetAdapterCount_graphicsvb
IDH_Direct3D8.GetAdapterDisplayMode_graphicsvb

Ordinal number that denotes the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter.

Mode

A **D3DDISPLAYMODE** type, to be filled with information describing the current adapter's mode.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Note that a device that represents an adapter may have been reset into a mode other than the adapter's mode. In this case, if the device is lost, it will not determine the adapter's current mode. This function returns the adapter mode, not any particular device's desired mode.

Direct3D8.GetAdapterIdentifier

#Describes the physical display adapters present in the system when the **Direct3D8** object was created.

object.**GetAdapterIdentifier(** _
Adapter **As Long**, _
Flags **As Long**, _
Identifier **As D3DADAPTER_IDENTIFIER8)**

Parts

object

Object expression that resolves to a **Direct3D8** object.

Adapter

Ordinal number that denotes the display adapter. D3DADAPTER_DEFAULT is always the primary display adapter. The minimum value for this parameter is 0, and the maximum value for this parameter is one less than the value returned by **Direct3D8.GetAdapterCount**.

Flags

IDH_Direct3D8.GetAdapterIdentifier_graphicsvb

Parameter that is typically set to 0 (zero). However, you can specify the following value defined by the **CONST_D3DCONST** enumeration.

D3DENUM_NO_WHQL_LEVEL
Forces the **WHQLLevel** member of the **D3DADAPTER_IDENTIFIER8** type to be 0 (zero). In this case, no information is returned for the WHQL certification date. Setting this flag avoids the one- or two-second time penalty incurred to determine the WHQL certification date.

Identifier
A **D3DADAPTER_IDENTIFIER8** type to be filled with information describing this adapter.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3D8.GetAdapterCount

Direct3D8.GetAdapterModeCount

#Returns the number of display modes available on this adapter.

object.**GetAdapterModeCount(_**
Adapter **As Long) As Long**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter
Ordinal number that denotes the display adapter. D3DADAPTER_DEFAULT is always the primary display adapter.

IDH_Direct3D8.GetAdapterModeCount_graphicsvb

Return Values

This method returns the number of display modes on this adapter, or 0 (zero) if *Adapter* is greater than or equal to the number of adapters on the system.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3D8.EnumAdapterModes

Direct3D8.GetAdapterMonitor

#Returns the handle of the monitor associated with the Microsoft® Direct3D® object.

object.**GetAdapterMonitor(_**
Adapter **As Long) As Long**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter
Ordinal number that denotes the display adapter. D3DADAPTER_DEFAULT is always the primary display adapter.

Return Values

Handle of the monitor associated with the Direct3D object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_Direct3D8.GetAdapterMonitor_graphicsvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

As shown in the following code fragment, which illustrates how to obtain a handle to the monitor associated with a given device, use **Direct3DDevice8.GetDirect3D** to return the Direct3D enumerator from the device, and use **Direct3DDevice8.GetCreationParameters** to retrieve the value for *Adapter*.

```
Dim Device As Direct3DDevice8, D3D As Direct3DDevice8
Dim Parameters As D3DDEVICE_CREATION_PARAMETERS
Dim MonitorHandle As Long

' The following code fragment assumes that the above items have been properly created and
initialized
If ((Device.GetCreationParameters(Parameters) <= 0) And (Device.GetDirect3D(D3D) <= 0))
Then
    MonitorHandle = D3D.GetAdapterMonitor(Parameters.AdapterOrdinal)
Else
    ' Error handling code here
End If
```

See Also

Direct3DDevice8.GetCreationParts, **Direct3DDevice8.GetDirect3D**

Direct3D8.GetDeviceCaps

#Retrieves device-specific information about a device.

object.**GetDeviceCaps(** _
Adapter **As Long**, _
DeviceType **As CONST_D3DDEVTYPE**, _
Caps **As D3DCAPS8)**

Parts

object
Object expression that resolves to a **Direct3D8** object.

Adapter

IDH_Direct3D8.GetDeviceCaps_graphicsvb

Ordinal number that denotes the display adapter. D3DADAPTER_DEFAULT is always the primary display adapter.

DeviceType

Member of the **CONST_D3DDEVTYPE** enumeration. Denotes the device type.

Caps

A **D3DCAPS8** type to be filled with information describing the capabilities of the device.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_INVALIDDEVICE

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic Error Handling topic.

Remarks

The application should not assume the persistence of vertex processing capabilities across Microsoft® Direct3D® device objects. The particular capabilities that a physical device exposes may depend on parameters supplied to **Direct3D8.CreateDevice**. For example, the capabilities might yield different vertex processing capabilities before and after creating a Direct3D Device Object with hardware vertex processing enabled. For more information, see **D3DCAPS8**.

Direct3D8.RegisterSoftwareDevice

Registers a pluggable software device with Microsoft® Direct3D®.

object.**RegisterSoftwareDevice(_**
InitializeFunction **As Any)**

Parts

object

Object expression that resolves to a **Direct3D8** object.

InitializeFunction

Initialization function for the software device to be registered.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Software rasterization for Direct3D is provided by pluggable software devices, enabling applications to access a variety of software rasterizers through the Direct3D objects. Software devices are loaded by the application and registered with the Direct3D object, at which point a Direct3DDevice object can be created that will perform rendering with the software device.

Direct3D software devices communicate with Direct3D through an interface similar to the hardware device driver interface (DDI).

The Direct3D DDK provides the documentation and headers for developing pluggable software devices.

Direct3DBaseTexture8

#Applications use the methods of the **Direct3DBaseTexture8** class to manipulate texture resources, including cube textures, textures and volume textures.

The **Direct3DBaseTexture8** class assigned to a particular stage for a device is obtained by calling the **Direct3DDevice8.GetTexture** method.

The **Direct3DBaseTexture8** class implements the following **Direct3DResource8** methods, which can be organized into these groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad

IDH_Direct3DBaseTexture8_graphicsvb

	SetPriority

The methods of the **Direct3DBaseTexture8** class can be organized into the following groups.

Detail	**GetLOD**
	SetLOD
Information	**GetLevelCount**

Direct3DBaseTexture8.GetLevelCount

#Returns the number of texture levels in a multilevel texture.

object.**GetLevelCount() As Long**

Parts

object
Object expression that resolves to a **Direct3DBaseTexture8** object.

Return Values

A **Long** value, indicating the number of texture levels in a multilevel texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DBaseTexture8**.

- **Direct3DCubeTexture8**
- **Direct3DTexture8**
- **Direct3DVolumeTexture8**

IDH_Direct3DBaseTexture8.GetLevelCount_graphicsvb

Direct3DBaseTexture8.GetLOD

#Returns a value clamped to the maximum level of detail (LOD) set for a managed texture.

object.**GetLOD() As Long**

Parts

object
Object expression that resolves to a **Direct3DBaseTexture8** object.

Return Values

A **Long** value, clamped to the maximum LOD value (one less than the total number of levels).

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DBaseTexture8**.

- **Direct3DCubeTexture8**
- **Direct3DTexture8**
- **Direct3DVolumeTexture8**

Remarks

GetLOD is used for LOD control of managed textures. This method returns 0 on nonmanaged textures.

See Also

Direct3DBaseTexture8.SetLOD

IDH_Direct3DBaseTexture8.GetLOD_graphicsvb

Direct3DBaseTexture8.SetLOD

#Sets the most detailed level of detail (LOD) for a managed texture.

object.**SetLOD(_**
LODNew **As Long) As Long**

Parts

object
Object expression that resolves to a **Direct3DBaseTexture8** object.

LODNew
Most detailed LOD value to set for the mipmap chain.

Return Values

A **Long** value, clamped to the maximum LOD value (one less than the total number of levels). Subsequent calls to this method return the clamped value, not the LOD value that was previously set.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DBaseTexture8**.

- **Direct3DCubeTexture8**
- **Direct3DTexture8**
- **Direct3DVolumeTexture8**

Remarks

SetLOD is used for LOD control of managed textures. This method returns 0 on nonmanaged textures.

SetLOD communicates to the Microsoft® Direct3D® texture manager the most detailed mipmap in the chain that should be loaded into local video memory. For example, in a five-level mipmap chain, setting *LODNew* to 2 indicates that the texture

IDH_Direct3DBaseTexture8.SetLOD_graphicsvb

manager should load only mipmap levels 2 through 4 into local video memory at any given time.

More specifically, if the texture was created with the dimensions of 256×256, setting the most detailed level to 0 indicates that 256×256 is the largest mipmap available, setting the most detailed level to 1 indicates that 128×128 is the largest mipmap available, and so on, up to the most detailed mip level (the smallest texture size) for the chain.

See Also

Direct3DBaseTexture8.GetLOD

Direct3DCubeTexture8

#Applications use the methods of the **Direct3DCubeTexture8** class to manipulate a cube map resource.

The **Direct3DCubeTexture8** class is obtained by calling the **Direct3DDevice8.CreateCubeTexture** method.

The **Direct3DCubeTexture8** class implements the following **Direct3DResource8** methods, which can be organized into the following groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The **Direct3DCubeTexture8** class implements the following **Direct3DBaseTexture8** class methods, which can be organized into the following groups.

Detail	**GetLOD**
	SetLOD
Information	**GetLevelCount**

The methods of the **Direct3DCubeTexture8** class can be organized into the following groups.

IDH_Direct3DCubeTexture8_graphicsvb

Information	**GetLevelDesc**
Locking Surfaces	**LockRect**
	UnlockRect
Miscellaneous	**AddDirtyRect**
	GetCubeMapSurface

See Also

Direct3DDevice8.CreateCubeTexture

Direct3DCubeTexture8.AddDirtyRect

#Adds a dirty region to a cube map resource.

object.**AddDirtyRect(_**
FaceType **As CONST_D3DCUBEMAP_FACES**, **_**
DirtyRect **As Any)**

Parts

object
Object expression that resolves to a **Direct3DCubeTexture8** object.

FaceType
Member of the **CONST_D3DCUBEMAP_FACES** enumeration, identifying the cube map face.

DirtyRect
A **RECT** type, specifying the dirty region. Specifying ByVal 0 expands the dirty region to cover the entire cube texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DCubeTexture8.GetLevelDesc, **Direct3DCubeTexture8.LockRect**, **Direct3DCubeTexture8.UnlockRect**

IDH_Direct3DCubeTexture8.AddDirtyRect_graphicsvb

Direct3DCubeTexture8.GetCubeMapSurface

#Retrieves a cube map surface.

object.**GetCubeMapSurface(_**
FaceType **As CONST_D3DCUBEMAP_FACES, _**
Level **As Long) As Direct3DSurface8**

Parts

object
Object expression that resolves to a **Direct3DCubeTexture8** object.

FaceType
Member of the **CONST_D3DCUBEMAP_FACES** enumeration, identifying a cube map face.

Level
Specifies a level of a mipmapped cube texture.

Return Values

This method returns a **Direct3DSurface8** object, representing the returned cube map surface.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DCubeTexture8.GetLevelDesc

#Retrieves a description of the specified cube texture level

object.**GetLevelDesc(_**
Level **As Long, _**
Desc **As D3DSURFACE_DESC)**

Parts

object

IDH_Direct3DCubeTexture8.GetCubeMapSurface_graphicsvb
IDH_Direct3DCubeTexture8.GetLevelDesc_graphicsvb

Object expression that resolves to a **Direct3DCubeTexture8** object.

Level
Specifies a level of a mipmapped cube texture.

Desc
A **D3DSURFACE_DESC** type, describing the specified cube texture level.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if one or more of the parameters are invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DCubeTexture8.AddDirtyRect, **Direct3DCubeTexture8.LockRect**, **Direct3DCubeTexture8.UnlockRect**

Direct3DCubeTexture8.LockRect

#Locks a rectangle on a cube texture resource.

```
object.LockRect( _
    FaceType As CONST_D3DCUBEMAP_FACES, _
    Level As Long, _
    LockedRect As D3DLOCKED_RECT, _
    Rect As Any, _
    Flags As Long)
```

Parts

object
Object expression that resolves to a **Direct3DCubeTexture8** object.

FaceType
Member of the **CONST_D3DCUBEMAP_FACES** enumeration, identifying a cube map face.

Level
Specifies a level of a mipmapped cube texture.

LockedRect
A **D3DLOCKED_RECT** type, describing the region to lock.

Rect
Rectangle to lock. Specified by a **RECT** type, or ByVal 0 to expand the dirty region to cover the entire texture.

IDH_Direct3DCubeTexture8.LockRect_graphicsvb

Flags
A combination of zero or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing the type of lock to perform.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

A multisample backbuffer cannot be locked.

See Also

Direct3DCubeTexture8.AddDirtyRect, **Direct3DCubeTexture8.GetLevelDesc**, **Direct3DCubeTexture8.UnlockRect**

Direct3DCubeTexture8.UnlockRect

#Unlocks a rectangle on a cube texture resource.

object.**UnlockRect(** _
FaceType **As CONST_D3DCUBEMAP_FACES**, _
Level **As Long)**

Parts

object
Object expression that resolves to a **Direct3DCubeTexture8** object.

FaceType
Member of the **CONST_D3DCUBEMAP_FACES** enumeration, identifying a cube map face.

Level
Specifies a level of a mipmapped cube texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

IDH_Direct3DCubeTexture8.UnlockRect_graphicsvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DCubeTexture8.AddDirtyRect, **Direct3DCubeTexture8.LockRect**

Direct3DDevice8

#Applications use the methods of the **Direct3DDevice8** class to perform DrawPrimitive-based rendering, create resources, work with system-level variables, adjust gamma ramp levels, and work with palettes.

The **Direct3DDevice8** class is obtained by calling the **Direct3D8.CreateDevice** method.

The methods of the **Direct3DDevice8** class can be organized into the following groups.

Cursors	**SetCursorPosition**
	SetCursorProperties
	ShowCursor
Creation	**CreateAdditionalSwapChain**
	CreateCubeTexture
	CreateDepthStencilSurface
	CreateImageSurface
	CreateIndexBuffer
	CreateRenderTarget
	CreateTexture
	CreateVertexBuffer
	CreateVolumeTexture
Device States	**ApplyStateBlock**
	BeginStateBlock
	CaptureStateBlock
	CreateStateBlock
	DeleteStateBlock
	EndStateBlock

IDH_Direct3DDevice8_graphicsvb

	GetClipStatus
	GetRenderState
	GetRenderTarget
	GetTransform
	SetClipStatus
	SetRenderState
	SetRenderTarget
	SetTransform
Gamma Ramps	**GetGammaRamp**
	SetGammaRamp
High-Order Primitives	**DeletePatch**
	DrawRectPatch
	DrawTriPatch
Index Data	**GetIndices**
	SetIndices
Information	**GetAvailableTextureMem**
	GetCreationParameters
	GetDeviceCaps
	GetDirect3D
	GetDisplayMode
	GetInfo
	GetRasterStatus
Lighting and Materials	**GetLight**
	GetLightEnable
	GetMaterial
	LightEnable
	SetLight
	SetMaterial
Miscellaneous	**CopyRects**
	MultiplyTransform
	ProcessVertices

	ResourceManagerDiscardBytes
	TestCooperativeLevel
Palettes	**GetCurrentTexturePalette**
	GetPaletteEntries
	SetCurrentTexturePalette
	SetPaletteEntries
Pixel Shaders	**CreatePixelShader**
	DeletePixelShader
	GetPixelShader
	GetPixelShaderConstant
	GetPixelShaderFunction
	SetPixelShader
	SetPixelShaderConstant
Presentation	**Present**
	Reset
Rendering	**DrawIndexedPrimitive**
	DrawIndexedPrimitiveUP
	DrawPrimitive
	DrawPrimitiveUP
Scenes	**BeginScene**
	EndScene
Stream Data	**GetStreamSource**
	SetStreamSource
Surfaces	**GetBackBuffer**
	GetDepthStencilSurface
	GetFrontBuffer
Textures	**GetTexture**
	GetTextureStageState
	SetTexture
	SetTextureStageState
	UpdateTexture

	ValidateDevice
User-Defined Clip Planes	**GetClipPlane**
	SetClipPlane
Viewports	**Clear**
	GetViewport
	SetViewport
Vertex Shaders	**CreateVertexShader**
	DeleteVertexShader
	GetVertexShader
	GetVertexShaderConstant
	GetVertexShaderDeclaration
	GetVertexShaderFunction
	SetVertexShader
	SetVertexShaderConstant

See Also

Direct3D8.CreateDevice

Direct3DDevice8.ApplyStateBlock

#Applies an existing device-state block to the rendering device.

object.**ApplyStateBlock(** _
Token **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Token
Handle to the device-state block to execute, as returned by a previous call to the **Direct3DDevice8.EndStateBlock** method.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

IDH_Direct3DDevice8.ApplyStateBlock_graphicsvb

Err.Number may be set to D3DERR_INVALIDCALL to indicate that the *Token* parameter is invalid or a macro is currently being recorded.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The value &hFFFFFFFF& is an invalid state block handle.

Applications cannot apply a device-state block while recording another block.

As with all operations that affect the state of the rendering device, it is recommended that you apply state blocks during scene rendering—that is, after calling the **Direct3DDevice8.BeginScene** method and before calling **Direct3DDevice8.EndScene**.

See Also

Direct3DDevice8.BeginStateBlock, **Direct3DDevice8.EndStateBlock**, **Direct3DDevice8.CaptureStateBlock**, **Direct3DDevice8.CreateStateBlock**, **Direct3DDevice8.DeleteStateBlock**

Direct3DDevice8.BeginScene

#Begins a scene.

object.**BeginScene()**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Applications must call this method before performing any rendering and must call **Direct3DDevice8.EndScene** when rendering is complete and before calling **BeginScene** again.

IDH_Direct3DDevice8.BeginScene_graphicsvb

If the **BeginScene** method fails, the device was unable to begin the scene, and there is no need to call the **Direct3DDevice8.EndScene** method. In fact, calls to **EndScene** fail if the previous call to **BeginScene** failed.

See Also

Direct3DDevice8.EndScene

Direct3DDevice8.BeginStateBlock

#Signals Microsoft® Direct3D® to begin recording a device-state block.

object.**BeginStateBlock()**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL
E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Applications can ensure that all recorded states are valid by calling the **Direct3DDevice8.ValidateDevice** method prior to calling this method.

The following methods can be recorded in a state block, after calling **BeginStateBlock** and before **Direct3DDevice8.EndStateBlock**.

- **Direct3DDevice8.LightEnable**
- **Direct3DDevice8.SetClipPlane**
- **Direct3DDevice8.SetIndices**
- **Direct3DDevice8.SetLight**
- **Direct3DDevice8.SetMaterial**
- **Direct3DDevice8.SetPixelShader**
- **Direct3DDevice8.SetPixelShaderConstant**

IDH_Direct3DDevice8.BeginStateBlock_graphicsvb

- **Direct3DDevice8.SetRenderState**
- **Direct3DDevice8.SetStreamSource**
- **Direct3DDevice8.SetTexture**
- **Direct3DDevice8.SetTextureStageState**
- **Direct3DDevice8.SetTransform**
- **Direct3DDevice8.SetViewport**
- **Direct3DDevice8.SetVertexShader**
- **Direct3DDevice8.SetVertexShaderConstant**

The ordering of state changes in a state block is not guaranteed. If the same state is specified multiple times in a state block, only the last value is used.

See Also

Direct3DDevice8.ApplyStateBlock, **Direct3DDevice8.EndStateBlock**, **Direct3DDevice8.CaptureStateBlock**, **Direct3DDevice8.CreateStateBlock**, **Direct3DDevice8.DeleteStateBlock**

Direct3DDevice8.CaptureStateBlock

#Updates the values within an existing state block to the values set for the device.

object.**CaptureStateBlock(** _
Token **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Token
Handle to the state block into which the device state is captured.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The value &hFFFFFFFF& is an invalid state block handle.

IDH_Direct3DDevice8.CaptureStateBlock_graphicsvb

This method captures updated values for states within an existing state block. It does not capture the entire state of the device.

CaptureStateBlock does not capture information for lights that are explicitly or implicitly created after the stateblock is created. For example, capturing the current state into a stateblock of type D3DSBT_ALL does not store information for lights that are created post-capture.

See Also

Direct3DDevice8.ApplyStateBlock, **Direct3DDevice8.BeginStateBlock**, **Direct3DDevice8.CreateStateBlock**, **Direct3DDevice8.EndStateBlock**, **Direct3DDevice8.DeleteStateBlock**

Direct3DDevice8.Clear

#Clears the viewport, or a set of rectangles in the viewport, to a specified RGBA color, clears the depth buffer, and erases the stencil buffer.

object.**Clear(** _
Count **As Long**, _
ClearD3DRect **As Any**,
Flags **As CONST_D3DCLEARFLAGS**, _
Color **As Long**, _
Z **As Single**, _
Stencil **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Count
Number of rectangles in the array at *ClearD3DRect*. If you set *ClearD3DRect* to ByVal 0, this parameter must be set to 0.

ClearD3DRect
First element of an array of **D3DRECT** types that describe the rectangles to clear. Set a rectangle to the dimensions of the rendering target to clear the entire surface. Each rectangle uses screen coordinates that correspond to points on the render target surface. Coordinates are clipped to the bounds of the viewport rectangle. This parameter can be set to ByVal 0 to indicate that the entire viewport rectangle is to be cleared.

Flags
A combination of the flags defined by the **CONST_D3DCLEARFLAGS** enumeration that indicate which surfaces should be cleared. Note that at least one flag must be used.

IDH_Direct3DDevice8.Clear_graphicsvb

Color
A 32-bit ARGB color value to which the render target surface is cleared.

Z
New z value that this method stores in the depth buffer. This parameter can be in the range from 0.0 through 1.0 (for z-based or w-based depth buffers). A value of 0.0 represents the nearest distance to the viewer, and 1.0 the farthest distance.

Stencil
Integer value to store in each stencil-buffer entry. This parameter can be in the range from 0 through 2^n–1, where *n* is the bit depth of the stencil buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method fails if you specify the D3DCLEAR_ZBUFFER or D3DCLEAR_STENCIL flags when the render target does not have an attached depth buffer. Similarly, if you specify the D3DCLEAR_STENCIL flag when the depth-buffer format does not contain stencil buffer information, this method fails.

The following code fragment shows how to call the default case for **Clear**.

```
Dim Device As Direct3DDevice8

' The following assumes that Flags, Color, Z, and Stencil
' have been set to valid values.
Device.Clear 0, ByVal 0, Flags, Color, Z, Stencil
```

The following code fragment show how to call **Clear** when specifying an array of rectangles to clear.

```
Dim clearD3dRects(10)
device.Clear 10, clearD3dRects(0), Flags, Color, Z, Stencil
```

Note that you pass in only the first element of the array of rectangles to clear.

Direct3DDevice8.CopyRects

#Copies rectangular subsets of pixels from one surface to another.

```
object.CopyRects( _
    SourceSurface As Direct3DSurface8, _
```

IDH_Direct3DDevice8.CopyRects_graphicsvb

FirstElementOfSourceRectsArray **As Any**, _
NumberOfRects **As Long**, _
DestinationSurface **As Direct3DSurface8**, _
FirstElementofDestPointArray **As Any)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

SourceSurface
A **Direct3DSurface8** object, representing the source surface. This parameter must be a different surface object than *DestinationSurface*.

FirstElementOfSourceRectsArray
First element of an array representing the rectangles to be transferred, or ByVal 0 to specify that the entire surface is copied. Each rectangle is transferred to the destination surface, with its top-left pixel at the position identified by the corresponding element of *FirstElementofDestPointArray*.

NumberOfRects
Number of rectangles contained in *FirstElementOfSourceRectsArray*.

DestinationSurface
A **Direct3DSurface8** object, representing the destination surface. This parameter must be a different surface object than *SourceSurface*.

FirstElementofDestPointArray
First element of an array that defines a series of points, identifying the top-left pixel position of each rectangle contained in *FirstElementOfSourceRectsArray*. If this parameter is ByVal 0, the rectangles are copied to the same offset (same top/left location) as the source rectangle.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method does not support stretch, color key, alpha blend, format conversion or clipping of either source or destination rectangles. Note that this method will fail unless all the source rectangles and their corresponding destination rectangles are completely contained within the source and destination surfaces respectively. The format of the two surfaces must match, but they can have different dimensions.

This method cannot be applied to surfaces whose formats are classified as depth stencil formats.

Direct3DDevice8.CreateAdditionalSwapChain

#Creates an additional swap chain for rendering multiple views.

object.**CreateAdditionalSwapChain(_**
PresentationParameters **As D3DPRESENT_PARAMETERS) As Direct3DSwapChain8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PresentationParameters
D3DPRESENT_PARAMETERS type, representing the presentation parameters for the new swap chain.

Return Values

An **Direct3DSwapChain8** object, representing the additional swap chain.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

There is always at least one swap chain for each device because Microsoft® Direct3D® for Microsoft DirectX® 8.0 has one swap chain as a property of the device.

Note that any given device can support only one full-screen swap chain.

See Also

Presenting Multiple Views in Windowed Mode

IDH_Direct3DDevice8.CreateAdditionalSwapChain_graphicsvb

Direct3DDevice8.CreateCubeTexture

#Creates a cube texture resource.

object.**CreateCubeTexture(_**
EdgeLength **As Long, _**
Levels **As Long, _**
Usage **As Long, _**
Format **As CONST_D3DFORMAT, _**
Pool **As CONST_D3DPOOL) As Direct3DCubeTexture8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

EdgeLength
Size of the edges of all the top-level faces of the cube texture. The pixel dimensions of subsequent levels of each face are the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. If the division by 2 results in 0, 1 will be taken instead.

Levels
The number of levels in each face of the cube texture. If this is 0 (zero), Microsoft® Direct3D® will generate all cube texture sublevels down to 1×1 pixels for each face for hardware that supports mipmapped cube textures. Otherwise, it will create one level. Call **Direct3DBaseTexture8.GetLevelCount** to see the number of levels generated.

Usage
A combination of one or more of the following flags defined by the **CONST_D3DUSAGEFLAGS** enumeration, describing the usage for this resource.

D3DUSAGE_DEPTHSTENCIL
Set to indicate that the surface is to be used as a depth-stencil surface. The resource can be passed to the *NewDepthStencil* parameter of the **Direct3DDevice8.SetRenderTarget** method. See Remarks.

D3DUSAGE_RENDERTARGET
Set to indicate that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method. See Remarks.

If either D3DUSAGE_RENDERTARGET or D3DUSAGE_DEPTHSTENCIL is specified, the application should check that the device supports these operations by calling **Direct3D8.CheckDeviceFormat**.

Format
Member of the **CONST_D3DFORMAT** enumeration, describing the format of all levels in all faces of the cube texture.

IDH_Direct3DDevice8.CreateCubeTexture_graphicsvb

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the cube texture should be placed.

Return Values

A **Direct3DCubeTexture8** object, representing the created cube texture resource.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **Direct3DCubeTexture8.GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

A texture (mipmap) is a collection of successively downsampled (mipmapped) surfaces. On the other hand, a cube texture (created by **CreateCubeTexture**) is a collection of six textures (mipmaps), one for each face. All faces must be present in the cube texture. Also, a cube map surface must be the same pixel size in all three dimensions (x, y, and z).

In DirectX 8.0, resource usage is enforced. An application that wishes to use a resource in a certain operation must specify that operation at resource creation time.

Direct3DDevice8.CreateDepthStencilSurface

#Creates a depth-stencil resource.

```
object.CreateDepthStencilSurface( _
    Width As Long, _
    Height As Long, _
    Format As CONST_D3DFORMAT, _
```

IDH_Direct3DDevice8.CreateDepthStencilSurface_graphicsvb

MultiSample **As CONST_D3DMULTISAMPLE_TYPE**, _
Surface **As Direct3DSurface8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Width
Width of the depth-stencil surface, in pixels.

Height
Height of the depth-stencil surface, in pixels.

Format
Member of the **CONST_D3DFORMAT** enumeration, describing the format of the depth-stencil surface. This value must be one of the enumerated depth-stencil formats for this device.

MultiSample
Member of the **CONST_D3DMULTISAMPLE_TYPE** enumeration, describing the multisampling buffer type. This value must be one of the allowed multisample types. When this surface is passed to **SetRenderTarget**, its multisample type must be the same as that of the render target.

Surface
A **Direct3DSurface8** object, representing the created depth-stencil surface resource.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The memory class of the depth-stencil buffer is always D3DPOOL_DEFAULT.

See Also

Direct3DDevice8.CopyRects, **Direct3DDevice8.SetRenderTarget**

Direct3DDevice8.CreateImageSurface

#Creates an image surface.

object.**CreateImageSurface(_**
Width **As Long, _**
Height **As Long, _**
Format **As CONST_D3DFORMAT) As Direct3DSurface8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Width
Width of the image surface, in pixels.

Height
Height of the image surface, in pixels.

Format
Member of the **CONST_D3DFORMAT** enumerated type, describing the format of the image surface.

Return Values

A **Direct3DSurface8** object, representing the created image surface.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Image surfaces are place holders, they are surfaces that cannot be used in any Microsoft® Direct3D® operations except locking and **Direct3DDevice8.CopyRects**.

Image surfaces are placed in the D3DPOOL_SYSTEMMEM memory class.

IDH_Direct3DDevice8.CreateImageSurface_graphicsvb

Direct3DDevice8.CreateIndexBuffer

#Creates an index buffer.

object.**CreateIndexBuffer(_**
LengthInBytes **As Long, _**
Usage **As Long, _**
Format **As CONST_D3DFORMAT, _**
Pool **As CONST_D3DPOOL) As Direct3DIndexBuffer8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

LengthInBytes
Size of the index buffer, in bytes.

Usage
A combination of one or more of the following flags defined by the **CONST_D3DUSAGEFLAGS** enumeration, describing the usage controls for this resource.

D3DUSAGE_DONOTCLIP
Set to indicate that the index buffer content will never require clipping.

D3DUSAGE_DYNAMIC
Set to indicate when the vertex or index buffer requires dynamic memory usage. This usage is useful for drivers because it enables them to decide where to place the driver. In general, static vertex buffers are placed in video memory and dynamic vertex buffers are placed in AGP memory. Note that there is no separate static usage; if you do not specify D3DUSAGE_DYNAMIC the vertex buffer is made static. D3DUSAGE_DYNAMIC is strictly enforced through the D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE locking flags. As a result, D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE are valid only on vertex and index buffers created with D3DUSAGE_DYNAMIC; they are not valid flags on static vertex buffers.

Note that D3DUSAGE_DYNAMIC cannot be specified on managed vertex and index buffers. For more information, see Managing Resources.

D3DUSAGE_RTPATCHES
Set to indicate when the index buffer is to be used for drawing high-order primitives.

D3DUSAGE_NPATCHES
Set to indicate when the index buffer is to be used for drawing N patches.

D3DUSAGE_POINTS

IDH_Direct3DDevice8.CreateIndexBuffer_graphicsvb

Set to indicate when the index buffer is to be used for drawing point sprites or indexed point lists.

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the buffer is to be used with software processing.

D3DUSAGE_WRITEONLY
Informs the system that the application writes only to the index buffer. Using this flag enables the driver to select the best memory location for efficient write operations and rendering. Attempts to read from an index buffer that is created with this capability can result in degraded performance.

Format
Member of the **CONST_D3DFORMAT** enumeration, describing the format of the index buffer. The valid settings are the following:

D3DFMT_INDEX16
Indices are 16 bits each.

D3DFMT_INDEX32
Indices are 32 bits each. See Remarks.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing a valid memory class into which to place the resource.

Return Values

A **Direct3DIndexBuffer8** object, representing the created index buffer resource.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Index buffers are memory resources used to hold indices, they are similar to both surfaces and vertex buffers. The use of index buffers enables Microsoft® Direct3D® to avoid unnecessary data copying and to place the buffer in the optimal memory type for the expected usage.

To use index buffers, create an index buffer, lock it, fill it with indices, unlock it, pass it to **Direct3DDevice8.SetIndices**, set up the vertices, set up the vertex shader, and call **Direct3DDevice8.DrawIndexedPrimitive** for rendering.

The **MaxVertexIndex** member of the **D3DCAPS8** type indicates the types of index buffers that are valid for rendering.

See Also

Direct3DIndexBuffer8.GetDesc

Direct3DDevice8.CreatePixelShader

#Creates a pixel shader.

object.**CreatePixelShader(_**
FunctionTokenArray **As Long) As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

FunctionTokenArray
First element of the pixel shader function token array, specifying the blending operations.

Return Values

Returned pixel shader handle.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.CreatePixelShader_graphicsvb

See Also

Direct3DDevice8.DeletePixelShader, **D3DX8.AssembleShader**, **D3DX8.AssembleShaderFromFile**

Direct3DDevice8.CreateRenderTarget

#Creates a render target surface.

object.**CreateRenderTarget(** _
Width **As Long**, _
Height **As Long**, _
Format **As CONST_D3DFORMAT**, _
MultiSample **As CONST_D3DMULTISAMPLE_TYPE**, _
Lockable **As Long) As Direct3DSurface8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Width
Width of the render target surface, in pixels.

Height
Height of the render target surface, in pixels.

Format
Member of the **CONST_D3DFORMAT** enumeration, describing the format of the render target.

MultiSample
Member of the **CONST_D3DMULTISAMPLE_TYPE** enumeration, describing the multisampling buffer type. This parameter specifies the anti-aliasing type for this render target. The type must be the same as that of the depth stencil buffer when both surfaces are passed to **SetRenderTarget**.

Lockable
Value indicating whether the render target surface is lockable. If set to True, the render target is lockable. If set to False, it is unlockable. Note that lockable render targets incur a performance cost on some graphics hardware.

Return Values

A **Direct3DSurface8** object, representing the render target surface.

Remarks

Render target surfaces are placed in the D3DPOOL_DEFAULT memory class.

IDH_Direct3DDevice8.CreateRenderTarget_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DDevice8.CreateStateBlock

#Creates a new state block that contains the values for all device states, vertex-related states, or pixel-related states.

object.**CreateStateBlock(_**
BlockType **As CONST_D3DSTATEBLOCKTYPE) As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

BlockType
Type of state data that the method should capture. This parameter can be set to a value defined in the **CONST_D3DSTATEBLOCKTYPE** enumeration.

Return Values

A **Long** value containing the state block handle if the method succeeds.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.CreateStateBlock_graphicsvb

Remarks

The value &hFFFFFFFF& is an invalid state block handle.

Vertex-related device states typically refer to those states that affect how the system processes vertices. Pixel-related states generally refer to device states that affect how the system processes pixel or depth-buffer data during rasterization. Some states are contained in both groups.

See Also

Direct3DDevice8.ApplyStateBlock, **Direct3DDevice8.BeginStateBlock**, **Direct3DDevice8.CaptureStateBlock**, **Direct3DDevice8.EndStateBlock**, **Direct3DDevice8.DeleteStateBlock**

Direct3DDevice8.CreateTexture

#Creates a texture resource.

object.**CreateTexture(** _
Width **As Long**, _
Height **As Long**, _
Levels **As Long**, _
Usage **As Long**, _
Format **As CONST_D3DFORMAT**, _
Pool **As CONST_D3DPOOL) As Direct3DTexture8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Width
Width of the top-level of the texture, in pixels. The pixel dimensions of subsequent levels of each face will be the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. If the division by 2 results in 0, 1 will be taken instead.

Height
Height of the top-level of the texture level, in pixels. The pixel dimensions of subsequent levels of each face will be the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. If the division by 2 results in 0, 1 will be taken instead.

Levels
The number of levels in the texture. If this is 0 (zero), Microsoft® Direct3D® will generate all texture sub-levels down to 1×1 pixels for hardware that supports

IDH_Direct3DDevice8.CreateTexture_graphicsvb

MIP mapped textures. Otherwise, it will create one level. Call **Direct3DBaseTexture8.GetLevelCount** to see the number of levels generated.

Usage

A combination of one or more of the following flags defined by the **CONST_D3DUSAGEFLAGS** enumeration, describing the usage for this resource.

D3DUSAGE_DEPTHSTENCIL

Set to indicate that the surface is to be used as a depth-stencil surface. The resource can be passed to the *NewDepthStencil* parameter of the **Direct3DDevice8.SetRenderTarget** method.

D3DUSAGE_RENDERTARGET

Set to indicate that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method.

If either D3DUSAGE_RENDERTARGET or D3DUSAGE_DEPTHSTENCIL is specified, the application should check that the device supports these operations by calling **Direct3D8.CheckDeviceFormat**.

Format

Member of the **CONST_D3DFORMAT** enumeration, describing the format of all levels in the texture.

Pool

Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the texture should be placed.

Return Values

A **Direct3DTexture8** object, representing the created texture resource.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

In order to call **Direct3DDevice8.SetRenderTarget** with a texture, you must select a level using **Direct3DTexture8.GetSurfaceLevel** and pass the resulting surface to **SetRenderTarget**.

Direct3DDevice8.CreateVertexBuffer

[#]Creates a vertex buffer.

object.**CreateVertexBuffer(_**
LengthInBytes **As Long, _**
Usage **As Long, _**
FVF **As Long, _**
Pool **As CONST_D3DPOOL) As Direct3DVertexBuffer8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

LengthInBytes
Size of the vertex buffer, in bytes. For Flexible Vertex Format (FVF) buffers, *Length* must be large enough to contain at least one vertex, but it need not be a multiple of the vertex size. *Length* is validated only for FVF buffers.

Usage
A combination of one or more of the following flags defined by the **CONST_D3DUSAGEFLAGS** enumeration, describing the usage controls for this resource.

D3DUSAGE_DONOTCLIP
Set to indicate that the vertex buffer content will never require clipping. When rendering with buffers that have this flag set, the D3DRS_CLIPPING renderstate must be set to False.

D3DUSAGE_DYNAMIC
Set to indicate when the vertex or index buffer requires dynamic memory usage. This usage is useful for drivers because it enables them to decide where to place the driver. In general, static vertex buffers will be placed in video memory and dynamic vertex buffers will be placed in AGP memory. Note that there is no separate static usage; if you do not specify D3DUSAGE_DYNAMIC the vertex buffer is made static. D3DUSAGE_DYNAMIC is strictly enforced through the D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE locking flags. As a result, D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE are valid only on vertex and index buffers created with D3DUSAGE_DYNAMIC; they are not valid flags on static vertex buffers.

For more information about using dynamic vertex buffers, see Using Dynamic Vertex and Index Buffers.

Note that D3DUSAGE_DYNAMIC cannot be specified on managed vertex and index buffers. For more information, see Managing Resources.

D3DUSAGE_RTPATCHES

[#] IDH_Direct3DDevice8.CreateVertexBuffer_graphicsvb

Set to indicate when the vertex buffer is to be used for drawing high-order primitives.

D3DUSAGE_NPATCHES
Set to indicate when the vertex buffer is to be used for drawing N patches.

D3DUSAGE_POINTS
Set to indicate when the vertex buffer is to be used for drawing point sprites or indexed point lists.

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the vertex buffer is to be used with software vertex processing.

D3DUSAGE_WRITEONLY
Informs the system that the application writes only to the vertex buffer. Using this flag enables the driver to choose the best memory location for efficient write operations and rendering. Attempts to read from a vertex buffer that is created with this capability will fail.

FVF
Combination of flexible vertex format flags, a usage specifier that describes the vertex format of the vertices in this buffer. When this parameter is set to a valid FVF code, the created vertex buffer is an FVF vertex buffer (see Remarks). Otherwise, when this parameter is set to 0 (zero), the vertex buffer is a non-FVF vertex buffer.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing a valid memory class into which to place the resource.

Return Values

A **Direct3DVertexBuffer8** object, representing the created vertex buffer resource.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **Direct3DDevice8** class supports rendering of primitives using vertex data stored in vertex buffer objects. Vertex buffers are created from the **Direct3DDevice8** class, and are usable only with the Direct3DDevice8 object from which they are created.

When set to a non-zero value, which must be a valid FVF code, the *FVF* parameter indicates that the buffer content is to be characterized by an FVF code. A vertex buffer that is created with an FVF code is referred to as an FVF vertex buffer. For more information, see FVF Vertex Buffers.

The D3DUSAGE_SOFTWAREPROCESSING flag specifies that the vertex buffer is to be used with software vertex processing. For more information, see Device Types and Vertex Processing Requirements.

Non-FVF buffers can be used to interleave data during multipass rendering or multitexture rendering in a single pass. To do this, put geometry data in one buffer and texture coordinate data for each texture to be rendered in separate buffers. When rendering, the buffer containing the geometry data is interleaved with each of the buffers containing the texture coordinates. If FVF buffers were used instead, each of them would need to contain identical geometry data in addition to the texture coordinate data specific to each texture rendered. This would result in either a speed or memory penalty, depending on the strategy used. For more information on texture coordinates, see Understanding Texture Coordinates.

See Also

Direct3DVertexBuffer8.GetDesc, **Direct3DDevice8.ProcessVertices**

Direct3DDevice8.CreateVertexShader

#Creates a vertex shader and if created successfully sets that shader as the current shader.

object.**CreateVertexShader(** _
DeclarationTokenArray **As Long**, _
FunctionTokenArray **As Any**, _
retHandle **As Long**, _
Usage **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

DeclarationTokenArray

IDH_Direct3DDevice8.CreateVertexShader_graphicsvb

First element of the vertex shader declaration token array. This parameter defines the inputs to the shader, including how the vertex elements within the input data streams are used by the shader.

FunctionTokenArray

First element of the vertex shader function token array. This parameter defines the operations to apply to each vertex. If this parameter is set to ByVal 0, a shader is created for the fixed-function pipeline and the parameter declaration indicated by *DeclarationTokenArray* is made current and available to be set in a subsequent call to **Direct3DDevice8.SetVertexShader**.

If this parameter is not set to ByVal 0, the shader is programmable.

retHandle

Returned vertex shader handle.

Usage

Usage controls for the vertex shader. The following flag can be set.

D3DUSAGE_SOFTWAREPROCESSING

Set to indicate that the vertex shader is to be used with software vertex processing. The D3DUSAGE_SOFTWAREPROCESSING flag must be set for vertex shaders used when the D3DRS_SOFTWAREVERTEXPROCESSING member of the **CONST_D3DRENDERSTATETYPE** enumerated type is True, and removed for vertex shaders used when D3DRS_SOFTWAREVERTEXPROCESSING is FALSE.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

A vertex shader is defined by two token arrays that specify the declaration and function of the shader. The token arrays are composed of single or multiple **Long** tokens terminated by a special &HFFFFFFFF token value.

The shader declaration defines the static external interface of the shader, including binding of stream data to vertex register inputs and values loaded into the shader constant memory. The shader function defines the operation of the shader as an array of instructions that are executed in order for each vertex processed during the time the

shader is bound to a device. Shaders created without a function array apply the fixed function vertex processing when that shader is current.

See d3dshader.bas for a definition of the constants used to generate the declaration token array.

See Also

Direct3DDevice8.DeleteVertexShader, **D3DX8.AssembleShader**, **D3DX8.AssembleShaderFromFile**, **D3DX8.DeclaratorFromFVF**

Direct3DDevice8.CreateVolumeTexture

#Creates a volume texture resource.

object.**CreateVolumeTexture(_**
Width **As Long**, _
Height **As Long**, _
Depth **As Long**, _
Levels **As Long**, _
Usage **As Long**, _
Format **As CONST_D3DFORMAT**, _
Pool **As CONST_D3DPOOL) As Direct3DVolumeTexture8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Width
Width of the top-level of the volume texture, in pixels. This value must be a power of two. The pixel dimensions of subsequent levels will be the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0, 1 will be taken instead.

Height
Height of the top-level of the volume texture, in pixels. This value must be a power of two. The pixel dimensions of subsequent levels will be the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0, 1 will be taken instead.

Depth
Depth of the top-level of the volume texture, in pixels. This value must be a power of two. The pixel dimensions of subsequent levels will be the truncated value of half of the previous level's pixel dimension (independently). Each

IDH_Direct3DDevice8.CreateVolumeTexture_graphicsvb

dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0, 1 will be taken instead.

Levels
The number of levels in the volume texture. If this is 0 (zero), Microsoft® Direct3D® will generate all texture sub-levels down to 1×1×1 pixels for hardware that supports mipmapped volume textures. Otherwise, it will create one level. Call **Direct3DBaseTexture8.GetLevelCount** to see the number of levels generated.

Usage
Currently not used, set to 0.

Format
Member of the **CONST_D3DFORMAT** enumeration, describing the format of all levels in the volume texture.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the volume texture should be placed.

Return Values

A **Direct3DVolumeTexture8** object, representing the created volume texture resource.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DDevice8.DeletePatch

#Frees a cached high-order patch.

object.**DeletePatch(** _
Handle **As Long)**

Parts

object

IDH_Direct3DDevice8.DeletePatch_graphicsvb

Object expression that resolves to a **Direct3DDevice8** object.

Handle
Handle of the cached high-order patch to delete.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL to indicate that the *Handle* parameter is invalid or a macro is currently being recorded.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.DrawRectPatch, **Direct3DDevice8.DrawTriPatch**, Drawing Patches

Direct3DDevice8.DeletePixelShader

#Deletes the pixel shader referred to by the specified handle.

object.**DeletePixelShader(_**
PixelShaderHandle **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PixelShaderHandle
Pixel shader handle, identifying the pixel shader to be deleted from its internal entry.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.DeletePixelShader_graphicsvb

See Also

Direct3DDevice8.CreatePixelShader

Direct3DDevice8.DeleteStateBlock

#Deletes a previously recorded device-state block.

object.**DeleteStateBlock(_**
Token **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Token
Handle to the device-state block to delete, as returned by a previous call to the **Direct3DDevice8.EndStateBlock** method.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

Err.Number may be set to D3DERR_INVALIDCALL to indicate that the *Token* parameter is invalid or a macro is currently being recorded.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The value &hFFFFFFFF& is an invalid state block handle.

Applications cannot delete a device-state block while another is being recorded

See Also

Direct3DDevice8.ApplyStateBlock, **Direct3DDevice8.BeginStateBlock**, **Direct3DDevice8.CaptureStateBlock**, **Direct3DDevice8.CreateStateBlock**, **Direct3DDevice8.EndStateBlock**

IDH_Direct3DDevice8.DeleteStateBlock_graphicsvb

Direct3DDevice8.DeleteVertexShader

#Deletes the vertex shader referred to by the specified handle and frees up the associated resources.

object.**DeleteVertexShader(_**
VertexShaderHandle **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.
VertexShaderHandle
Vertex shader handle, identifying the vertex shader to be deleted.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.CreateVertexShader

Direct3DDevice8.DrawIndexedPrimitive

#Renders the specified geometric primitive, based on indexing into an array of vertices.

object.**DrawIndexedPrimitive(_**
PrimitiveType **As CONST_D3DPRIMITIVETYPE**, _
MinIndex **As Long**, _
NumIndices **As Long**, _
StartIndex **As Long**, _
PrimitiveCount **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

IDH_Direct3DDevice8.DeleteVertexShader_graphicsvb
IDH_Direct3DDevice8.DrawIndexedPrimitive_graphicsvb

PrimitiveType
Member of the **CONST_D3DPRIMITIVETYPE** enumeration, describing the type of primitive to render. See Remarks

MinIndex
Minimum vertex index for vertices used during this call.

NumIndices
The number of indices used during this call starting from *BaseVertexIndex* + *MinIndex*.

StartIndex
Location in the index array to start reading indices.

PrimitiveCount
Number of primitives to render. The number of indices used is a function of the primitive count and the primitive type. The maximum number of primitives allowed is determined by checking the *MaxPrimitiveCount* member of the **D3DCAPS8** structure.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method draws indexed primitives from the current set of data input streams.

MinIndex and all the indices in the index stream are relative to the *BaseVertexIndex*, set during the **Direct3DDevice8.SetIndices** call.

The *MinIndex* and *NumIndices* parameters specify the range of vertex indices used for each **DrawIndexedPrimitive** call. These are used to optimize vertex processing of indexed primitives by processing a sequential range of vertices prior to indexing into these vertices. It is invalid for any indices used during this call to reference any vertices outside of this range.

DrawIndexedPrimitive fails if no index array is set.

The D3DPT_POINTLIST member of the **CONST_D3DPRIMITIVETYPE** enumeration is not supported and should not be specified for *PrimitiveType*.

See Also

Direct3DDevice8.DrawPrimitive, **Direct3DDevice8.SetStreamSource**, Rendering Primitives

Direct3DDevice8.DrawIndexedPrimitiveUP

#Renders the specified geometric primitive with data specified by a user memory array.

object.**DrawIndexedPrimitiveUP(_**
PrimitiveType **As CONST_D3DPRIMITIVETYPE**, _
MinIndex **As Long**, _
NumVertices **As Long**, _
PrimitiveCount **As Long**, _
IndexDataArray **As Any**, _
IndexDataFormat **As CONST_D3DFORMAT**, _
VertexStreamZeroDataArray **As Any**, _
VertexStreamZeroStride **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PrimitiveType
Member of the **CONST_D3DPRIMITIVETYPE** enumeration, describing the type of primitive to render.

MinIndex
Minimum vertex index, relative to 0 (zero) (the start of *IndexDataArray*), for vertices used during this call.

NumVertices
The number of vertices used during this call, starting from *MinIndex*.

PrimitiveCount
Number of primitives to render. The number of indices used is a function of the primitive count and the primitive type. The maximum number of primitives allowed is determined by checking the *MaxPrimitiveCount* member of the **D3DCAPS8** structure.

IndexDataArray
First element of the user memory array containing the index data.

IndexDataFormat
Member of the **CONST_D3DFORMAT** enumeration, describing the format of the index data. The valid settings are the following:

D3DFMT_INDEX16
Indices are 16 bits each.

D3DFMT_INDEX32
Indices are 32 bits each.

IDH_Direct3DDevice8.DrawIndexedPrimitiveUP_graphicsvb

VertexStreamZeroDataArray
First element of the user memory array containing the vertex data to use for vertex stream zero.

VertexStreamZeroStride
Stride between data for each vertex, in bytes, as returned by the **Len** Microsoft® Visual Basic® function.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method is intended for use in applications that are unable to store their vertex data in vertex buffers. This method supports only a single vertex stream. The effect of this call is to use the provided vertex data pointer and stride for vertex stream zero. It is invalid to have the declaration of the current vertex shader refer to vertex streams other than stream zero.

Following any **DrawIndexedPrimitiveUP** call, the stream zero settings, referenced by **Direct3DDevice8.GetStreamSource**, are set to ByVal 0. Also, the index buffer setting for **Direct3DDevice8.SetIndices** is set to ByVal 0.

See Also

Direct3DDevice8.DrawPrimitiveUP, **Direct3DDevice8.SetStreamSource**, Rendering Primitives

Direct3DDevice8.DrawPrimitive

#Renders nonindexed primitives from the current set of data input streams.

```
object.DrawPrimitive( _
    PrimitiveType As CONST_D3DPRIMITIVETYPE, _
    StartVertex As Long, _
    PrimitiveCount As Long)
```

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PrimitiveType

IDH_Direct3DDevice8.DrawPrimitive_graphicsvb

Member of the **CONST_D3DPRIMITIVETYPE** enumeration, describing the type of primitive to render.

StartVertex
Index of the first vertex to load. Beginning at *StartVertex*, the correct number of vertices will be read out of the vertex buffer.

PrimitiveCount
Number of primitives to render. The maximum number of primitives allowed is determined by checking the *MaxPrimitiveCount* member of the **D3DCAPS8** structure. *PrimitiveCount* is the number of primitives as determined by the primitive type. If it is a line list, each primitive has two vertices. If it is a triangle list, each primitive has three vertices.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

DrawPrimitive should not be called with a single triangle at a time.

See Also

Direct3DDevice8.DrawIndexedPrimitive, **Direct3DDevice8.SetStreamSource**, Rendering Primitives

Direct3DDevice8.DrawPrimitiveUP

#Renders data specified by a user memory pointer as a sequence of geometric primitives of the specified type.

```
object.DrawPrimitiveUP( _
    PrimitiveType As CONST_D3DPRIMITIVETYPE, _
    PrimitiveCount As Long, _
    VertexStreamZeroDataArray As Any, _
    VertexStreamZeroStride As Long)
```

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PrimitiveType

IDH_Direct3DDevice8.DrawPrimitiveUP_graphicsvb

Member of the **CONST_D3DPRIMITIVETYPE** enumeration, describing the type of primitive to render.

PrimitiveCount
Number of primitives to render. The maximum number of primitives allowed is determined by checking the *MaxPrimitiveCount* member of the **D3DCAPS8** structure.

VertexStreamZeroDataArray
First element of the user memory array containing the vertex data to use for vertex stream zero.

VertexStreamZeroStride
Stride between data for each vertex, in bytes. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method is intended for use in applications that are unable to store their vertex data in vertex buffers. This method supports only a single vertex stream. The effect of this call is to use the provided vertex data pointer and stride for vertex stream zero. It is invalid to have the declaration of the current vertex shader refer to vertex streams other than stream zero.

The following code fragment shows how to determine *VertexStreamZeroStride*, using the Visual Basic **Len** function.

```
Dim device As Direct3DDevice8
Dim verts(3) As D3DVERTEX

device.DrawPrimitiveUP D3DPT_TRIANGLELIST, 1, verts(0), Len(vert(0))
```

Note that you pass in only the first element of the array containing the vertex data.

Following any **DrawPrimitiveUP** call, the stream zero settings, referenced by **Direct3DDevice8.GetStreamSource**, are set to ByVal 0.

See Also

Direct3DDevice8.DrawIndexedPrimitiveUP, **Direct3DDevice8.SetStreamSource**, Rendering Primitives

Direct3DDevice8.DrawRectPatch

#Draws a rectangular high-order patch using the currently set streams.

object.**DrawRectPatch(** _
Handle **As Long**, _
NumSegments **As Single**, _
Surface **As Any)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Handle
Handle to the rectangular high-order patch to draw.

NumSegments
Number of segments that each edge of the high-order primitive should be divided into when tessellated.

Surface
D3DRECTPATCH_INFO type, describing the rectangular high-order patch to draw.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.DeletePatch, Drawing Patches

Direct3DDevice8.DrawTriPatch

#Draws a triangular high-order patch using the currently set streams.

object.**DrawTriPatch(** _
Handle **As Long**, _
NumSegments **As Single**, _
Surface **As Any)**

IDH_Direct3DDevice8.DrawRectPatch_graphicsvb
IDH_Direct3DDevice8.DrawTriPatch_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Handle
Handle to the triangular high-order patch to draw.

NumSegments
Number of segments that each edge of the high-order primitive should be divided into when tessellated.

Surface
D3DTRIPATCH_INFO type, describing the triangular high-order patch to draw.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.DeletePatch, Drawing Patches

Direct3DDevice8.EndScene

#Ends a scene that was begun by calling the **Direct3DDevice8.BeginScene** method.

object.**EndScene()**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.EndScene_graphicsvb

Remarks

When this method succeeds, the scene has been rendered and the device surface holds the rendered scene.

When scene rendering begins successfully, you must call this method before you can call the **Direct3DDevice8.BeginScene** method to start rendering another scene. If a prior call to **BeginScene** method fails, the scene did not begin and this method should not be called.

See Also

Direct3DDevice8.BeginScene

Direct3DDevice8.EndStateBlock

#Signals Microsoft® Direct3D® to stop recording a device-state block and retrieve a handle to the state block.

object.**EndStateBlock() As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Return Values

Variable to be filled with the handle to the completed device state block. This value is used with the **Direct3DDevice8.ApplyStateBlock** and **Direct3DDevice8.DeleteStateBlock** methods.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL to indicate that the return value is invalid or that the **Direct3DDevice8.BeginStateBlock** method has not been called.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.ApplyStateBlock, **Direct3DDevice8.BeginStateBlock**, **Direct3DDevice8.CaptureStateBlock**, **Direct3DDevice8.CreateStateBlock**, **Direct3DDevice8.DeleteStateBlock**

IDH_Direct3DDevice8.EndStateBlock_graphicsvb

Direct3DDevice8.GetAvailableTextureMem

#Returns an estimate of the amount of available texture memory.

object.**GetAvailableTextureMem(_**
Pool **As CONST_D3DPOOL) As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Pool
CONST_D3DPOOL type, specifying the pool type to check.

Return Values

The function returns an estimate of the available texture memory.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The returned value is rounded to the nearest megabyte (MB). This is done to reflect the fact that video memory estimates are never precise due to alignment and other issues that affect consumption by certain resources. Applications can use this value to make gross estimates of memory availability to make large-scale resource decisions such as how many levels of a mipmap to attempt to allocate. However, applications cannot use this value to make small-scale decisions such as if there is enough memory left to allocate another resource.

Direct3DDevice8.GetBackBuffer

#Retrieves a back buffer from the device's swap chain.

object.**GetBackBuffer(_**
BackBuffer **As Long**, **_**
BufferType **As CONST_D3DBACKBUFFERTYPE) As Direct3DSurface8**

IDH_Direct3DDevice8.GetAvailableTextureMem_graphicsvb
IDH_Direct3DDevice8.GetBackBuffer_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

BackBuffer
Index of the back buffer object to return. Back buffers are numbered from 0 to the total number of back buffers - 1. A value of 0 returns the first back buffer, not the front buffer. The front buffer is not accessible through this method.

BufferType
Stereo view is not supported in DirectX 8.0, so the only valid value for this parameter is D3DBACKBUFFER_TYPE_MONO.

Return Values

A **Direct3DSurface8** object, representing the returned back buffer surface.

Error Codes

If *BackBuffer* exceeds or equals the total number of back buffers, an error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DDevice8.GetClipPlane

#Retrieves the coefficients of a user-defined clipping plane for the device.

```
object.GetClipPlane( _
    Index As Long, _
    Plane As D3DPLANE)
```

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Index
Index of the clipping plane for which the plane equation coefficients are retrieved.

Plane
A **D3DPLANE** type containing the clipping plane coefficients to be set, in the form of the general plane equation. See Remarks.

IDH_Direct3DDevice8.GetClipPlane_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL. This error indicates that the value in *Index* exceeds the maximum clipping plane index supported by the device.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The coefficients that this method reports take the form of the general plane equation. The values in the **D3DPLANE** type *Plane*—a, b, c, and d—fit into the general plane equation so that ax + by + cz + d = 0. A point with homogeneous coordinates (x, y, z, w) is visible in the half space of the plane if ax+ by + cz + dw >= 0. Points that exist on or behind the clipping plane are clipped from the scene.

The plane equation used by this method exists in world space and is set by a previous call to the **Direct3DDevice8.SetClipPlane** method.

See Also

Direct3DDevice8.SetClipPlane, D3DPLANE

Direct3DDevice8.GetClipStatus

#Retrieves the clip status.

object.**GetClipStatus(_**
ClipStatus **As D3DCLIPSTATUS8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

ClipStatus
A **D3DCLIPSTATUS8** type that describes the clip status.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if the parameter is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.GetClipStatus_graphicsvb

See Also

Direct3DDevice8.SetClipStatus

Direct3DDevice8.GetCreationParameters

#Retrieves the creation parameters of the device.

object.**GetCreationParameters(_**
CreationParams **As D3DDEVICE_CREATION_PARAMETERS)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

CreationParams
A **D3DDEVICE_CREATION_PARAMETERS** type, describing the creation parameters of the device.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if the parameter is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

You can query the **AdapterOrdinal** member of the returned **D3DDEVICE_CREATION_PARAMETERS** type to retrieve the ordinal of the adapter represented by this device.

Direct3DDevice8.GetCurrentTexturePalette

#Retrieves the current texture palette.

object.**GetCurrentTexturePalette() As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

IDH_Direct3DDevice8.GetCreationParameters_graphicsvb
IDH_Direct3DDevice8.GetCurrentTexturePalette_graphicsvb

Return Values

Returns a value that identifies the current texture palette.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetCurrentTexturePalette, Texture Palettes

Direct3DDevice8.GetDepthStencilSurface

#Retrieves the depth-stencil surface owned by the **Direct3DDevice** object.

object.**GetDepthStencilSurface() As Direct3DSurface8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Return Values

A **Direct3DSurface8** object, representing the returned depth-stencil surface.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetRenderTarget, **Direct3DDevice8.CopyRects**

IDH_Direct3DDevice8.GetDepthStencilSurface_graphicsvb

Direct3DDevice8.GetDeviceCaps

#Retrieves the capabilities of the rendering device.

object.**GetDeviceCaps(_**
Caps **As D3DCAPS8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Caps
A **D3DCAPS8** type, describing the returned device.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

GetDeviceCaps retrieves the software vertex pipeline capabilities when the device is being used in software vertex processing mode. Software vertex processing mode is selected when a device has been created with D3DCREATE_SOFTWAREVERTEXPROCESSING, or when a device has been created with D3DCREATE_MIXEDVERTEXPROCESSING and D3DRS_SOFTWAREVERTEXPROCESSING is set to True.

Direct3DDevice8.GetDirect3D

#Returns a reference to the Microsoft® Direct3D® object that created the device.

object.**GetDirect3D() As Direct3D8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

IDH_Direct3DDevice8.GetDeviceCaps_graphicsvb
IDH_Direct3DDevice8.GetDirect3D_graphicsvb

Return Values

A **Direct3D8** object, representing the Direct3D object that created the device.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DDevice8.GetDisplayMode

#Retrieves the display mode's spatial resolution, color resolution, and refresh frequency.

object.**GetDisplayMode(_**
Mode **As D3DDISPLAYMODE)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Mode
A **D3DDISPLAYMODE** type containing data about the display mode of the adapter, as opposed to the display mode of the device, which may not be active if the device does not own full-screen mode.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DDevice8.GetFrontBuffer

#Generates a copy of the device's front buffer and places that copy in a system memory buffer provided by the application.

object.**GetFrontBuffer(_**
DestSurface **As Direct3DSurface8)**

IDH_Direct3DDevice8.GetDisplayMode_graphicsvb
IDH_Direct3DDevice8.GetFrontBuffer_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

DestSurface
A **Direct3DSurface8** object that will receive a copy of the contents of the front buffer. The data is returned in successive rows with no intervening space, starting from the vertically highest row on the device's output to the lowest.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_DEVICELOST

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The buffer returned in *DestSurface* will be filled with a representation of the front buffer, converted to the standard 32bpp format, D3DFMT_A8R8G8B8.

Direct3DDevice8.GetGammaRamp

#Retrieves the gamma correction ramp for the swap chain.

object.**GetGammaRamp(_**
Ramp **As D3DGAMMARAMP)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Ramp
An application-supplied **D3DGAMMARAMP** type to fill with the gamma correction ramp.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

IDH_Direct3DDevice8.GetGammaRamp_graphicsvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetGammaRamp, **Direct3D8.CreateDevice**

Direct3DDevice8.GetIndices

#Retrieves index data.

object.**GetIndices(** _
RetIndexData **As Direct3DIndexBuffer8**, _
RetBaseVertexIndex **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

RetIndexData
A **Direct3DIndexBuffer8** object, representing the returned index data.

RetBaseVertexIndex
A variable holding the returned base value for vertex indices. This value is added to all indices prior to referencing vertex data, defining a starting position in the vertex streams.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The values in the index list are used to index into a vertex list when creating geometry to render.

The value returned in the *RetBaseVertexIndex* parameter specifies the base value for indices. This base value is added to all indices prior to referencing into the vertex data streams, which results in setting a starting position in the vertex data streams. The base vertex index enables multiple indexed primitives to be packed into a single set of vertex data without requiring the indices to be recomputed based on where the corresponding primitive is placed in the vertex data.

IDH_Direct3DDevice8.GetIndices_graphicsvb

See Also

Direct3DDevice8.SetIndices

Direct3DDevice8.GetInfo

#Retrieves information about the rendering device.

```
object.GetInfo( _
    DevInfoID As Long, _
    Info As Any, _
    Size As Long)
```

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

DevInfoID
Value used to identify what information will be returned in *Info*.

Info
A type that receives the specified device information if the call succeeds.

Size
Size of the type at *Info*, in bytes, as returned by the **Len** Microsoft® Visual Basic® function.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Visual Basic Error Handling topic.

Remarks

Information returned by this method can pertain to Microsoft® Direct3D® or the underlying device driver. This method makes it possible for drivers to declare specific information types, and corresponding structures, that are not documented in this SDK.

This method executes synchronously, and it can reduce your application's performance when it executes slowly. Do not call this method during scene rendering (between calls to **Direct3DDevice8.BeginScene** and **Direct3DDevice8.EndScene**).

This method is intended to be used for performance tracking and debugging during product development (on the debug version of Microsoft DirectX®). It can succeed without retrieving device data. This occurs when the retail version of the DirectX runtime is installed on the host system.

IDH_Direct3DDevice8.GetInfo_graphicsvb

Direct3DDevice8.GetLight

#Retrieves a set of lighting properties that this device uses.

object.**GetLight(_**
Index **As Long**, **_**
Light **As D3DLIGHT8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Index
Zero-based index of the lighting property set to retrieve.

Light
A **D3DLIGHT8** type that is filled with the retrieved lighting-parameter set.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL if one of the parameters is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetLight, **Direct3DDevice8.GetLightEnable**, **Direct3DDevice8.LightEnable**

Direct3DDevice8.GetLightEnable

#Retrieves the activity status—enabled or disabled—for a set of lighting parameters within a device.

object.**GetLightEnable(_**
Index **As Long) As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Index

IDH_Direct3DDevice8.GetLight_graphicsvb
IDH_Direct3DDevice8.GetLightEnable_graphicsvb

Zero-based index of the set of lighting parameters that are the target of this method.

Return Values

A variable to fill with the status of the specified lighting parameters. After the call, a nonzero value at this address indicates that the specified lighting parameters are enabled; a value of 0 indicates that they are disabled.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.GetLight, **Direct3DDevice8.LightEnable**, **Direct3DDevice8.SetLight**

Direct3DDevice8.GetMaterial

#Retrieves the current material properties for the device.

object.**GetMaterial(** _
Material **As D3DMATERIAL8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Material
A **D3DMATERIAL8** type to fill with the currently set material properties.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL to indicate the *Material* parameter is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.GetMaterial_graphicsvb

See Also

Direct3DDevice8.SetMaterial

Direct3DDevice8.GetPaletteEntries

[#]Retrieves palette entries.

```
object.GetPaletteEntries( _
    PaletteNumber As Long, _
    ArrayOfEntries As Any)
```

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PaletteNumber
An ordinal value identifying the particular palette to retrieve.

ArrayOfEntries
A **PALETTEENTRY** type to be filled with the returned palette entries. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY**, see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

See Also

Direct3DDevice8.GetCurrentTexturePalette, **Direct3DDevice8.SetCurrentTexturePalette**, **Direct3DDevice8.SetPaletteEntries**, Texture Palettes

---

[#] IDH_Direct3DDevice8.GetPaletteEntries_graphicsvb

Direct3DDevice8.GetPixelShader

#Retrieves the currently set pixel shader.

object.**GetPixelShader() As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Return Values

Pixel shader handle, representing the returned the pixel shader.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetPixelShader

Direct3DDevice8.GetPixelShaderConstant

#Retrieves the values in the pixel constant array.

object.**GetPixelShaderConstant(** _
Register **As Long**, _
ConstantData **As Any**, _
ConstantCount **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Register
Register address at which to start retrieving data from the pixel constant array.

IDH_Direct3DDevice8.GetPixelShader_graphicsvb
IDH_Direct3DDevice8.GetPixelShaderConstant_graphicsvb

ConstantData
First element of an array to hold the retrieved values from the pixel constant array.

ConstantCount
Number of constants to retrieve from the pixel constant array. Each constant is comprised of four **Single** values.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This is the method used to retrieve the values in the constant registers of the pixel shader assembler.

See Also

Direct3DDevice8.SetPixelShaderConstant

Direct3DDevice8.GetPixelShaderFunction

#Retrieves the pixel shader function.

object.**GetPixelShaderFunction(** _
Handle **As Long**, _
Data **As Any**, _
SizeOfData **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Handle
Handle to the referred to pixel shader.

Data
First element of an array to be filled with the code associated with the requested pixel shader handle, if the call succeeds.

SizeOfData

IDH_Direct3DDevice8.GetPixelShaderFunction_graphicsvb

Size of the array at *Data*, in bytes. If this value is less than the actual size of the data (such as 0) the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA. If *Data* is ByVal 0, then this parameter is filled with the required size and D3D_OK is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The pixel shader function specifies blending operations.

See Also

Direct3DDevice8.GetPixelShader

Direct3DDevice8.GetRasterStatus

#Returns information describing the raster of the monitor on which the swap chain is presented.

object.**GetRasterStatus() As D3DRASTER_STATUS**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Return Values

A **D3DRASTER_STATUS** type filled with information about the position or other status of the raster on the monitor driven by this adapter.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if *pRasterStatus* is invalid or if the device does not support reading the current scan line. You can determine whether or not the device supports reading the scan line by checking for the D3DCAPS_READ_SCANLINE flag in the **Caps** member of **D3DCAPS8**.

IDH_Direct3DDevice8.GetRasterStatus_graphicsvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DRASTER_STATUS

Direct3DDevice8.GetRenderState

#Retrieves a render-state value for a device.

object.**GetRenderState(_**
State **As CONST_D3DRENDERSTATETYPE) As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

State
Device state variable that is being queried. This parameter can be any member of the **CONST_D3DRENDERSTATETYPE** enumeration.

Return Values

A variable that receives the value of the queried render state variable when the method returns.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL if one of the parameters is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetRenderState

Direct3DDevice8.GetRenderTarget

#Retrieves a pointer to the Microsoft® Direct3D® surface that is being used as a render target.

object.**GetRenderTarget() As Direct3DSurface8**

IDH_Direct3DDevice8.GetRenderState_graphicsvb
IDH_Direct3DDevice8.GetRenderTarget_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Return Values

A **Direct3DSurface8** object, representing the returned render target surface for this device.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL if one of the parameters is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetRenderTarget

Direct3DDevice8.GetStreamSource

#Retrieves a vertex buffer bound to the specified data stream.

object.**GetStreamSource(** _
StreamNumber **As Long**, _
RetStreamData **As Direct3DVertexBuffer8**, _
RetStride **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

StreamNumber
Specifies the data stream, in the range from 0 to the maximum number of streams - 1.

RetStreamData
A **Direct3DVertexBuffer8** object, representing the returned vertex buffer bound to the specified data stream.

RetStride
Returned stride of the component, in bytes. See Remarks.

IDH_Direct3DDevice8.GetStreamSource_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

A stream is defined as a uniform array of component data, where each component consists of one or more elements representing a single entity such as position, normal, color, and so on.

The following code fragment shows how to call **GetStreamSource**.

```
Dim device As Direct3DDevice8
Dim retVB As Direct3DVertexBuffer8
Dim retStride As Long

device.GetStreamSource 0, retVB, retStride
```

When **GetStreamSource** returns, *retVB* and *retStride* have been set to valid values.

See Also

Direct3DDevice8.SetStreamSource

Direct3DDevice8.GetTexture

#Retrieves a texture assigned to a stage for a device.

object.**GetTexture(_**
Stage **As Long) As Direct3DBaseTexture8**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Stage
Stage identifier of the texture to retrieve. Stage identifiers are zero-based. Devices can have up to eight set textures, so the maximum allowable value allowed for *Stage* is 7.

Return Values

A **Direct3DBaseTexture8** object, representing the returned texture.

IDH_Direct3DDevice8.GetTexture_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetTexture, **Direct3DDevice8.GetTextureStageState**, **Direct3DDevice8.SetTextureStageState**

Direct3DDevice8.GetTextureStageState

#Retrieves a state value for an assigned texture.

object.**GetTextureStageState(_**
Stage **As Long**, **_**
State **As CONST_D3DTEXTURESTAGESTATETYPE) As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Stage
Stage identifier of the texture for which the state is retrieved. Stage identifiers are zero-based. Devices can have up to eight set textures, so the maximum allowable value allowed for *Stage* is 7.

State
Texture state to retrieve. This parameter can be any member of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

Return Values

Returns the state value.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.GetTextureStageState_graphicsvb

See Also

Direct3DDevice8.SetTextureStageState, **Direct3DDevice8.GetTexture**, **Direct3DDevice8.SetTexture**

Direct3DDevice8.GetTransform

#Retrieves a matrix describing a transformation state.

object.**GetTransform(** _
TransformType **As CONST_D3DTRANSFORMSTATETYPE**, _
Matrix **As D3DMATRIX)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

TransformType
Device state variable that is being modified. This parameter can be any member of the **CONST_D3DTRANSFORMSTATETYPE** enumeration.

Matrix
A **D3DMATRIX** type, describing the returned transformation state.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetTransform

Direct3DDevice8.GetVertexShader

#Retrieves the currently set vertex shader.

object.**GetVertexShader() As Long**

Parts

object

IDH_Direct3DDevice8.GetTransform_graphicsvb
IDH_Direct3DDevice8.GetVertexShader_graphicsvb

Object expression that resolves to a **Direct3DDevice8** object.

Return Values

Vertex shader handle, representing the returned vertex shader.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if the Direct3DDevice object is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.SetVertexShader

Direct3DDevice8.GetVertexShaderConstant

#Retrieves the values in the vertex constant array.

object.**GetVertexShaderConstant(** _
Register **As Long**, _
ConstantData **As Any**, _
ConstantCount **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Register
Register address at which to start retrieving data from the vertex constant array.

ConstantData
First element of an array to hold the retrieved values from the vertex constant array.

ConstantCount
Number of constants to retrieve from the vertex constant array. Each constant is comprised of four **Single** values.

IDH_Direct3DDevice8.GetVertexShaderConstant_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This is the method used to retrieve the values in the constant registers of the vertex shader assembler.

See Also

Direct3DDevice8.SetVertexShaderConstant

Direct3DDevice8.GetVertexShaderDeclaration

#Retrieves the vertex shader declaration token array.

object.**GetVertexShaderDeclaration(_**
Handle **As Long**, **_**
Data **As Any**, **_**
SizeOfData **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Handle
Handle to the referred to vertex shader.

Data
First element of an array to be filled with the declaration associated with the requested vertex shader handle, if the call succeeds.

SizeOfData
Size of the array at *Data*, in bytes. If this value is less than the actual size of the data (such as 0) the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA. If *Data* is ByVal 0, then this parameter is filled with the required size and D3D_OK is returned.

IDH_Direct3DDevice8.GetVertexShaderDeclaration_graphicsvb

Error Codes

If *Handle* is an invalid handle to a vertex shader, D3DERR_INVALIDCALL is returned.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The vertex shader declaration token array defines the inputs to the shader, including how the vertex elements within the input data streams are used by the shader.

See Also

Direct3DDevice8.GetVertexShader

Direct3DDevice8.GetVertexShaderFunction

#Retrieves the vertex shader function.

object.**GetVertexShaderFunction(** _
Handle **As Long**, _
Data **As Any**, _
SizeOfData **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Handle
Handle to the referred to vertex shader.

Data
First element of an array to be filled with the code associated with the requested vertex shader handle, if the call succeeds.

SizeOfData
Size of the array at *Data*, in bytes. If this value is less than the actual size of the data (such as 0) the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA. If *Data* is ByVal 0, then this parameter is filled with the required size and D3D_OK is returned.

IDH_Direct3DDevice8.GetVertexShaderFunction_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_MOREDATA

If *Handle* is an invalid handle to a vertex shader, D3DERR_INVALIDCALL is returned.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The vertex shader function defines the operation to apply to each vertex.

See Also

Direct3DDevice8.GetVertexShader

Direct3DDevice8.GetViewport

#Retrieves the viewport parameters currently set for the device.

object.**GetViewport(_**
Viewport **As D3DVIEWPORT8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Viewport
A **D3DVIEWPORT8** type, representing the returned viewport parameters.

Error Codes

An error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL if the *Viewport* parameter is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.GetViewport_graphicsvb

See Also

Direct3DDevice8.SetViewport

Direct3DDevice8.LightEnable

#Enables or disables a set of lighting parameters within a device.

object.**LightEnable(** _
Index **As Long**, _
Enabled **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Index
Zero-based index of the set of lighting parameters that are the target of this method.

Enabled
Value that indicates if the set of lighting parameters are being enabled or disabled. Set this parameter True to enable lighting with the parameters at the specified index, or FALSE to disable it.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If you supply a value for *Index* outside the range of the light property sets assigned within the device, the **LightEnable** method creates a light source represented by a **D3DLIGHT8** type with the following properties, and it sets its enabled state to the value specified in *Enabled*.

Member	**Default**
Type	D3DLIGHT_DIRECTIONAL
Diffuse	(R:1, G:1, B:1, A:0)
Specular	(R:0, G:0, B:0, A:0)
Ambient	(R:0, G:0, B:0, A:0)

IDH_Direct3DDevice8.LightEnable_graphicsvb

Position	(0, 0, 0)
Direction	(0, 0, 1)
Range	0
Falloff	0
Attenuation0	0
Attenuation1	0
Attenuation2	0
Theta	0
Phi	0

See Also

Direct3DDevice8.GetLight, **Direct3DDevice8.GetLightEnable**, **Direct3DDevice8.SetLight**

Direct3DDevice8.MultiplyTransform

#Multiplies a device's world, view, or projection matrices by a specified matrix.

object.**MultiplyTransform(** _
TransformType **As CONST_D3DTRANSFORMSTATETYPE**, _
Matrix **As D3DMATRIX)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

TransformType
Member of the **CONST_D3DTRANSFORMSTATETYPE** enumeration that identifies which device matrix is to be modified. The most common setting, D3DTS_WORLD, modifies the world matrix, but you can specify that the method modify the view or projection matrices, if needed.

Matrix
A **D3DMATRIX** type that modifies the current transformation.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set. The method returns D3DERR_INVALIDCALL if one of the parameters is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.MultiplyTransform_graphicsvb

Remarks

An application might use the **MultiplyTransform** method to work with hierarchies of transformations. For example, the geometry and transformations describing an arm might be arranged in the following hierarchy:

```
shoulder_transformation
   upper_arm geometry
   elbow transformation
      lower_arm geometry
      wrist transformation
         hand geometry
```

An application might use the following series of calls to render this hierarchy. (Not all of the parameters are shown in this pseudocode.)

```
Direct3DDevice8.SetTransform(D3DTS_WORLD,
   shoulder_transform)
Direct3DDevice8.DrawPrimitive(upper_arm)
Direct3DDevice8.MultiplyTransform(D3DTS_WORLD,
   elbow_transform)
Direct3DDevice8.DrawPrimitive(lower_arm)
Direct3DDevice8.MultiplyTransform(D3DTS_WORLD,
   wrist_transform)
Direct3DDevice8.DrawPrimitive(hand)
```

See Also

Direct3DDevice8.DrawPrimitive, **Direct3DDevice8.SetTransform**

Direct3DDevice8.Present

[#]Presents the contents of the next in the sequence of back buffers owned by the device.

object.**Present(** _
   SourceRect **As Any**, _
   DestRect **As Any**, _
   DestWindowOverride **As Long**, _
   DirtyRegion **As Any)**

Parts

object
   Object expression that resolves to a **Direct3DDevice8** object.

IDH_Direct3DDevice8.Present_graphicsvb

SourceRect
Value that must be ByVal 0 unless the swap chain was created with D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC. *SourceRect* is a type containing the source rectangle. If ByVal 0, the entire source surface is presented. If the rectangle exceeds the source surface, the rectangle is clipped to the source surface.

DestRect
Value that must be ByVal 0 unless the swap chain was created with D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC. *DestRect* is a type containing the destination rectangle, in window client coordinates. If ByVal 0, the entire client area is filled. If the rectangle exceeds the destination client area, the rectangle is clipped to the destination client area.

DestWindowOverride
Destination window whose client area is taken as the target for this presentation. If this value is 0 (zero), then the **hWndDeviceWindow** member of **D3DPRESENT_PARAMETERS** is taken.

DirtyRegion
This parameter is not used and should be set to ByVal 0.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_DEVICELOST

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If necessary, a stretch operation is applied to transfer the pixels within the source rectangle to the destination rectangle in the client area of the target window.

The following code fragment shows how to call the default case for **Present**.

```
Dim device As Direct3DDevice8
device.Present ByVal 0, ByVal 0, 0, ByVal 0
```

For the default case, note that you must specify ByVal 0 for the parameters that expect type **Any**. If you specify 0 (zero) for a parameter of type **Any**, an error will be generated and the method will fail.

See Also

Direct3DDevice8.Reset

Direct3DDevice8.ProcessVertices

#Applies the vertex processing defined by the vertex shader to the set of input data streams, generating a single stream of interleaved vertex data to the destination vertex buffer.

object.**ProcessVertices(** _
SrcStartIndex **As Long**, _
DestIndex **As Long**, _
VertexCount **As Long**, _
DestBuffer **As Direct3DVertexBuffer8**, _
Flags **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

SrcStartIndex
Index of first vertex to be loaded.

DestIndex
Index of first vertex in the destination vertex buffer into which the results are placed.

VertexCount
Number of vertices to process.

DestBuffer
A **Direct3DVertexBuffer8** object, the destination vertex buffer representing the stream of interleaved vertex data.

Flags
Processing options. Set this parameter to 0 for default processing. Set to D3DPV_DONOTCOPYDATA to prevent the system from copying vertex data not affected by the vertex operation into the destination buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The destination vertex buffer, *DestBuffer*, must be created with a nonzero *FVF* parameter. The FVF code specified during the call to the

---

IDH_Direct3DDevice8.ProcessVertices_graphicsvb

Direct3DDevice8.CreateVertexBuffer method, specifies the vertex elements present in the destination vertex buffer.

See Also

Device Types and Vertex Processing Requirements

Direct3DDevice8.Reset

#Changes the type, size, and format of the swap chain.

object.**Reset(** _
PresentationParameters **As D3DPRESENT_PARAMETERS)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PresentationParameters
A **D3DPRESENT_PARAMETERS** type, describing the new presentation parameters.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If a call to **Reset** fails, the device will be placed in the "lost" state (as indicated by a return value of D3DERR_DEVICELOST from a call to **TestCooperativeLevel**) unless it is already in the "not reset" state (as indicated by a return value of D3DERR_DEVICENOTRESET from a call to **TestCooperativeLevel**). Refer to **Direct3DDevice8.TestCooperativeLevel** and Lost Devices for further information concerning the use of **Reset** in the context of lost devices.

Calling **Reset** causes all texture memory surfaces to be lost, managed textures to be flushed from video memory, and all state information to be lost. Before calling the

IDH_Direct3DDevice8.Reset_graphicsvb

Reset method for a device, an application should release any explicit render targets, depth stencil surfaces, additional swap chains and D3DPOOL_DEFAULT resources associated with the device.

The different types of swap chains are full-screen or windowed. If the new swap chain is full-screen, the adapter will be placed in the display mode that matches the new size.

When **Reset** fails, the only valid methods you can call are **Reset** and **Direct3DDevice8.TestCooperativeLevel**. Calling any other method may result in an exception.

See Also

CONST_D3DSWAPEFFECT, **D3DPRESENT_PARAMETERS**, **Direct3DDevice8.Present**

Direct3DDevice8.ResourceManagerDiscardBytes

#Invokes the resource manager to free memory.

object.**ResourceManagerDiscardBytes(** _
NumberOfBytes **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

NumberOfBytes
The number of target bytes to discard. If 0 (zero), then the resource manager should discard all bytes.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The resource manager frees up memory allocations until the number of target bytes is reached. The resource manager follows a least-recently-used (LRU) priority technique.

IDH_Direct3DDevice8.ResourceManagerDiscardBytes_graphicsvb

Note that the number of bytes freed are not guaranteed to be contiguous, so you might not be able to create a resource that takes the number of bytes specified in *Bytes* after calling this method.

Direct3DDevice8.SetClipPlane

#Sets the coefficients of a user-defined clipping plane for the device.

```
object.SetClipPlane( _
    Index As Long, _
    Plane As D3DPLANE)
```

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Index
Index of the clipping plane for which the plane equation coefficients are to be set.

Plane
A **D3DPLANE** type containing the clipping plane coefficients to be set, in the form of the general plane equation. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL. This error indicates that the value in *Index* exceeds the maximum clipping plane index supported by the device.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The coefficients that this method sets take the form of the general plane equation. The values in the **D3DPLANE** type *Plane*—a, b, c, and d—fit into the general plane equation so that ax + by + cz + d = 0. A point with homogeneous coordinates (x, y, z, w) is visible in the half space of the plane if ax+ by + cz + dw >= 0. Points that exist on or behind the clipping plane are clipped from the scene.

When the fixed function pipeline is used the plane equations are assumed to be in world space. When the programmable pipeline is used the plane equations are assumed to be in the clipping space (the same space as output vertices).

This method does not enable the clipping plane equation being set. To enable a clipping plane, use the **D3DRS_CLIPPLANEENABLE** render state constant of the **CONST_D3DRENDERSTATETYPE** enumeration.

IDH_Direct3DDevice8.SetClipPlane_graphicsvb

See Also

Direct3DDevice8.GetClipPlane

Direct3DDevice8.SetClipStatus

#Sets the clip status.

object.**SetClipStatus(_**
ClipStatus **As D3DCLIPSTATUS8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

ClipStatus
A **D3DCLIPSTATUS8** type, describing the clip status settings to be set.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if the *ClipStatus* parameter is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.GetClipStatus

Direct3DDevice8.SetCurrentTexturePalette

#Sets the current texture palette.

object.**SetCurrentTexturePalette(_**
PaletteNumber **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PaletteNumber
Value that specifies the texture palette to set as the current texture palette.

IDH_Direct3DDevice8.SetClipStatus_graphicsvb
IDH_Direct3DDevice8.SetCurrentTexturePalette_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

A single logical palette is associated with the device and is shared by all texture stages.

See Also

Direct3DDevice8.GetCurrentTexturePalette, Texture Palettes

Direct3DDevice8.SetCursorPosition

#Sets the cursor position and update options.

```
object.SetCursorPosition( _
    XScreenSpace As Long, _
    YScreenSpace As Long
    Flags As CONST_D3DSCPFLAGS)
```

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

XScreenSpace
The new X-position of the cursor in screen-space coordinates. See Remarks.

YScreenSpace
The new Y-position of the cursor in screen-space coordinates. See Remarks.

Flags
Specifies the update options for the cursor. Currently, only one flag is defined.

D3DCURSOR_IMMEDIATE_UPDATE

Update cursor at the refresh rate.

If this flag is specified, the system guarantees that the cursor will be updated at a minimum of half the display refresh rate but never more frequently than the display refresh rate. Otherwise, the method delays cursor updates until the next **Direct3DDevice8.Present** call. This default behavior usually results in better performance than if the flag had been set. However, applications should set this

IDH_Direct3DDevice8.SetCursorPosition_graphicsvb

flag if the rate of calls to **Present** is high enough that users would notice a significant delay in cursor motion.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

When running in full-screen mode, screen-space coordinates are the back buffer coordinates appropriately scaled to the current display mode. When running in windowed mode, screen-space coordinates are the desktop coordinates. The cursor image is drawn at the specified position minus the hotspot offset specified by the **Direct3DDevice8.SetCursorProperties** method.

If the cursor has been hidden by **Direct3DDevice8.ShowCursor**, then the cursor is not drawn.

See Also

Direct3DDevice8.SetCursorProperties, **Direct3DDevice8.ShowCursor**

Direct3DDevice8.SetCursorProperties

#Sets properties for the cursor.

object.**SetCursorProperties(** _
XHotSpot **As Long**, _
YHotSpot **As Long**, _
CursorSurface **As Direct3DSurface8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

XHotSpot
X-coordinate offset into the cursor, in pixels from the top-left corner, that is considered the center. When the cursor is given a new position, the image is drawn at an offset from this new position determined by subtracting the hot spot coordinates from the position.

YHotSpot

IDH_Direct3DDevice8.SetCursorProperties_graphicsvb

Y-coordinate offset into the cursor, in pixels from the top-left corner, that is considered the center. When the cursor is given a new position, the image is drawn at an offset from this new position determined by subtracting the hot spot coordinates from the position.

CursorSurface

A **Direct3DSurface8** object. This parameter must point to an 8888 ARGB surface (format D3DFORMAT_A8R8G8B8). The contents of this surface will be copied and potentially format-converted into an internal buffer from which the cursor is displayed. The dimensions of this surface must be less than the dimensions of the display mode, and must be a power of two in each direction, although not necessarily the same power of two. The alpha channel must be either 0.0 or 1.0.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method is independent of the Microsoft® Win32® API **ShowCursor** and **SetCursor** functions. Applications should either use the Win32 cursor or the Microsoft Direct3D® cursor, not both.

The application can determine what hardware support is available for cursors by examining appropriate members of the **D3DCAPS8** type. Typically, hardware supports only 32x32 cursors. Additionally, when windowed, the system might support only 32x32 cursors. In this case, **SetCursorProperties** still succeeds, but the cursor may be reduced to that size—the hot spot is scaled appropriately.

See Also

Direct3DDevice8.SetCursorPosition, **Direct3DDevice8.ShowCursor** , **D3DCAPS8**

Direct3DDevice8.SetGammaRamp

[#]Sets the gamma correction ramp for the implicit swap chain.

```
object.SetGammaRamp( _
    Flags As Long, _
    Ramp As D3DGAMMARAMP)
```

[#] IDH_Direct3DDevice8.SetGammaRamp_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Flags
Indicates whether correction should be applied. Gamma correction results in a more consistent display, but it can incur processing overhead and should not be used frequently. Short-duration effects such as flashing the whole screen red should not be calibrated, but long-duration gamma changes should be calibrated. One of the values defined by the **CONST_D3DSGRFLAGS** enumeration can be set.

Ramp
A **D3DGAMMARAMP** type, representing the gamma correction ramp to be set for the implicit swap chain.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

There is always at least one swap chain (the implicit swap chain) for each device because Microsoft® Direct3D® for Microsoft DirectX® 8.0 has one swap chain as a property of the device.

Because the gamma ramp is a property of the swap chain, the gamma ramp may be applied when the swap chain is windowed.

The gamma ramp takes effect immediately. No wait for VSYNC is performed.

If the device does not support gamma ramps in the swap chain's current presentation mode (full-screen or windowed), no error return is given. Applications can check the D3DCAPS2_FULLSCREENGAMMA and D3DCAPS2_CANCALIBRATEGAMMA capability bits in the **Caps2** member of the **D3DCAPS8** type to determine the capabilities of the device and whether a calibrator is installed.

See Also

Direct3D8.CreateDevice, **Direct3DDevice8.GetGammaRamp**

Direct3DDevice8.SetIndices

#Sets index data.

object.**SetIndices(** _
IndexData **As Direct3DIndexBuffer8**, _
BaseVertexIndex **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

IndexData
A **Direct3DIndexBuffer8** object, representing the index data to be set.

BaseVertexIndex
Base value for vertex indices. This value is added to all indices prior to referencing vertex data, defining a starting position in the vertex streams.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **SetIndices** method sets the current index array to an index buffer. The single set of indices is used to index all streams.

The *BaseVertexIndex* parameter specifies the base value for indices. This base value is added to all indices prior to referencing into the vertex data streams, which results in setting a starting position in the vertex data streams. The base vertex index enables multiple indexed primitives to be packed into a single set of vertex data without requiring the indices to be recomputed based on where the corresponding primitive is placed in the vertex data.

See Also

Direct3DDevice8.GetIndices

Direct3DDevice8.SetLight

#Assigns a set of lighting properties for this device.

IDH_Direct3DDevice8.SetIndices_graphicsvb
IDH_Direct3DDevice8.SetLight_graphicsvb

object.**SetLight(** _
Index **As Long**, _
Light **As D3DLIGHT8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Index
Zero-based index of the set of lighting properties to set. If a set of lighting properties exists at this index, it is overwritten by the new properties specified in *Light*.

Light
A **D3DLIGHT8** type, containing the lighting-parameters to set.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.GetLight, **Direct3DDevice8.GetLightEnable**, **Direct3DDevice8.LightEnable**

Direct3DDevice8.SetMaterial

#Sets the material properties for the device.

object.**SetMaterial(** _
Material **As D3DMATERIAL8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Material
A **D3DMATERIAL8** type, describing the material properties to set.

IDH_Direct3DDevice8.SetMaterial_graphicsvb

Error Codes

If the method fails, the return value is an error. The method returns D3DERR_INVALIDCALL if the *Material* parameter is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.GetMaterial

Direct3DDevice8.SetPaletteEntries

#

Sets palette entries.

object.**SetPaletteEntries(_**
PaletteNumber **As Long**, _
ArrayOfEntries **As Any)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PaletteNumber
An ordinal value identifying the particular palette upon which the operation is to be performed.

ArrayOfEntries
First element of an array containing the palette entries to set. The number of **PALETTEENTRY** types in this array is assumed to be 256. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY**, see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented

IDH_Direct3DDevice8.SetPaletteEntries_graphicsvb

in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

A single logical palette is associated with the device, and is shared by all texture stages.

The following code fragment shows how to call **SetPaletteEntries**.

```
Dim device As Direct3DDevice8
Dim pal(255) As PALETTEENTRY

'Pass the first element of the array of palette entries
device.SetPaletteEntries 255, pal(0)
```

Ensure that *PaletteNumber* matches the size of the passed array of palette entries.

See Also

Direct3DDevice8.GetCurrentTexturePalette, **Direct3DDevice8.GetPaletteEntries**, **Direct3DDevice8.SetCurrentTexturePalette**, Texture Palettes

Direct3DDevice8.SetPixelShader

#Sets the current pixel shader to a previously created pixel shader.

object.**SetPixelShader(** _
PixelShaderHandle **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

PixelShaderHandle
Handle to the pixel shader, specifying the pixel shader to set. The value for this parameter can be the handle returned by **Direct3DDevice8.CreatePixelShader**.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DDevice8.SetPixelShader_graphicsvb

See Also

Direct3DDevice8.GetPixelShader

Direct3DDevice8.SetPixelShaderConstant

#Sets the values in the pixel constant array.

object.**SetPixelShaderConstant(** _
Register **As Long**, _
ConstantData **As Any**, _
ConstantCount **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Register
Register address at which to start loading data into the pixel constant array.

ConstantData
First element of an array holding the values to load into the pixel constant array.

ConstantCount
Number of constants to load into the pixel constant array. Each constant is comprised of four **Single** values.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This is the method used to load the constant registers of the pixel shader assembler.

See Also

Direct3DDevice8.GetPixelShaderConstant

IDH_Direct3DDevice8.SetPixelShaderConstant_graphicsvb

Direct3DDevice8.SetRenderState

#Sets a single device render-state parameter.

object.**SetRenderState(_**
State **As CONST_D3DRENDERSTATETYPE, _**
Value **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

State
Device state variable that is being modified. This parameter can be any member of the **CONST_D3DRENDERSTATETYPE** enumeration.

Value
New value for the device render state to be set. The meaning of this parameter is dependent on the value specified for *State*. For example, if *State* were D3DRS_SHADEMODE, the second parameter would be one member of the **CONST_D3DSHADEMODE** enumeration.

Error Codes

If the method fails, it sets **Err.Number** to an error code and raises an error. The error code is D3DERR_INVALIDCALL if one of the parameters is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.GetRenderState, **Direct3DDevice8.SetTransform**

Direct3DDevice8.SetRenderTarget

#Sets a new color buffer, depth buffer, or both for the device.

object.**SetRenderTarget(_**
NewRenderTarget **As Direct3DSurface8, _**
NewDepthStencil **As Direct3DSurface8)**

Parts

object

IDH_Direct3DDevice8.SetRenderState_graphicsvb
IDH_Direct3DDevice8.SetRenderTarget_graphicsvb

Object expression that resolves to a **Direct3DDevice8** object.

NewRenderTarget

New color buffer. If Nothing, the existing color buffer is retained. If this parameter is not Nothing, the reference count on the new render target is incremented. Devices always have to be associated with a color buffer.

The new render target surface must have at least D3DUSAGE_RENDERTARGET specified.

NewDepthStencil

New depth-stencil buffer. If a depth-stencil buffer exists, it is released. If this parameter is not Nothing, the reference count on the new depth-stencil buffer surface is incremented. Applications can change the render target without changing the depth buffer by passing in the **Direct3DSurface8** object returned by **Direct3DDevice8.GetDepthStencilSurface**.

The new depth-stencil surface must have at least D3DUSAGE_DEPTHSTENCIL and D3DPOOL_DEFAULT specified.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL if *NewRenderTarget* or *NewDepthStencil* are not Nothing and invalid, or if the new depth buffer is smaller than the new or retained color buffer.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Once a color buffer and a depth-stencil surface have been associated with the same device by this method, they are said to be paired.

The previous depth-stencil surface's contents persist after a call to **SetRenderTarget** to disassociate the previous depth-stencil surface from the device. If the surface is re-associated with the device, then the contents of the surface will be unchanged, providing the color buffer to which the new depth-stencil surface is being paired is the same size and format as the color buffer to which the depth-stencil surface was most recently paired.

See Also

Direct3DDevice8.CreateDepthStencilSurface, **Direct3DDevice8.GetDepthStencilSurface**

Direct3DDevice8.SetStreamSource

#Binds a vertex buffer to a device data stream.

IDH_Direct3DDevice8.SetStreamSource_graphicsvb

object.**SetStreamSource(** _
StreamNumber **As Long**, _
StreamData **As Direct3DVertexBuffer8**, _
Stride **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

StreamNumber
Specifies the data stream, in the range from 0 to the maximum number of streams - 1.

StreamData
A **Direct3DVertexBuffer8** object, representing the vertex buffer to bind to the specified data stream.

Stride
Stride of the component, in bytes. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

When a flexible vertex format (FVF) vertex shader is used, the stream vertex stride in **SetStreamSource** must match the vertex size, computed from the FVF. When a declaration is used, the stream vertex stride in **SetStreamSource** should be greater than or equal to the stream size, computed from the declaration.

The **SetStreamSource** method binds a vertex buffer to a device data stream. For details, see Setting the Stream Source.

See Also

Direct3DDevice8.DrawIndexedPrimitive, **Direct3DDevice8.DrawIndexedPrimitiveUP**, **Direct3DDevice8.DrawPrimitive**, **Direct3DDevice8.DrawPrimitiveUP**, **Direct3DDevice8.GetStreamSource**

Direct3DDevice8.SetTexture

#Assigns a texture to a stage for a device.

IDH_Direct3DDevice8.SetTexture_graphicsvb

object.**SetTexture(** _
Stage **As Long**, _
Texture **As Direct3DBaseTexture8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Stage
Stage identifier to which the texture is set. Stage identifiers are zero-based. Devices can have up to eight set textures, so the maximum allowable value allowed for *Stage* is 7.

Texture
A **Direct3DBaseTexture8** object, representing the texture being set. For complex textures, such as mipmaps and cube textures, this parameter must be the top-level surface.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method increments the reference count of the texture surface being assigned, and it decrements the reference count of the previously selected texture if there is one. When the texture is no longer needed, set the texture at the appropriate stage to Nothing. Failure to do this results in a memory leak.

See Also

Direct3DDevice8.GetTexture, **Direct3DDevice8.GetTextureStageState**, **Direct3DDevice8.SetTextureStageState**

Direct3DDevice8.SetTextureStageState

#Sets the state value for the currently assigned texture.

object.**SetTextureStageState(** _
Stage **As Long**, _
Type **As CONST_D3DTEXTURESTAGESTATETYPE**, _
Value **As Long)**

IDH_Direct3DDevice8.SetTextureStageState_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Stage
Stage identifier of the texture for which the state value is set. Stage identifiers are zero-based. Devices can have up to eight set textures, so the maximum allowable value allowed for *Stage* is 7.

Type
Texture state to set. This parameter can be any member of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

Value
State value to set. The meaning of this value is determined by the *Type* parameter.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.GetTextureStageState, **Direct3DDevice8.GetTexture**, **Direct3DDevice8.SetTexture**

Direct3DDevice8.SetTransform

#Sets a single device transformation-related state.

object.**SetTransform(** _
TransformType **As CONST_D3DTRANSFORMSTATETYPE**, _
Matrix **As D3DMATRIX)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

TransformType
Device-state variable that is being modified. This parameter can be any member of the **CONST_D3DTRANSFORMSTATETYPE** enumeration.

IDH_Direct3DDevice8.SetTransform_graphicsvb

Matrix
A **D3DMATRIX** type that modifies the current transformation.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if one of the parameters is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DDevice8.GetTransform, **Direct3DDevice8.SetRenderState**

Direct3DDevice8.SetVertexShader

#Sets the current vertex shader to a previously created vertex shader or to a flexible vertex format (FVF) fixed function shader.

object.**SetVertexShader(_**
VertexShaderHandle **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

VertexShaderHandle
Handle to a vertex shader, specifying the vertex shader to set. The value for this parameter can be the handle returned by **Direct3DDevice8.CreateVertexShader**, or an FVF code. An FVF code is a combination of flexible vertex format flags.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The effect of using an FVF code in place of the handle is to enable a fixed-function vertex shader, with an implicit declaration that matches the FVF code contents read

IDH_Direct3DDevice8.SetVertexShader_graphicsvb

from stream zero. Only stream zero is referenced when an FVF-specified shader is bound to the device.

See Also

Direct3DDevice8.GetVertexShader

Direct3DDevice8.SetVertexShaderConstant

#Sets values in the vertex constant array.

object.**SetVertexShaderConstant(_**
Register **As Long**, _
ConstantData **As Any**, _
ConstantCount **As Long)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Register
Register address at which to start loading data into the vertex constant array.

ConstantData
First element of an array holding the values to load into the vertex constant array.

ConstantCount
Number of constants to load into the vertex constant array. Each constant is comprised of four **Single** values.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This is the method used to load the constant registers of the vertex shader assembler. When loading transformation matrices, the application should transpose them row/column and load them into consecutive constant registers.

IDH_Direct3DDevice8.SetVertexShaderConstant_graphicsvb

See Also

Direct3DDevice8.GetVertexShaderConstant

Direct3DDevice8.SetViewport

#Sets the viewport parameters for the device.

object.**SetViewport(_**
Viewport **As D3DVIEWPORT8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Viewport
A **D3DVIEWPORT8** type, specifying the viewport parameters to set.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL to indicate that the *Viewport* parameter is invalid.

If the viewport parameters described by the **D3DVIEWPORT8** type describe a region that cannot exist within the render target surface, the method fails, returning D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Any call to **SetViewPort** between a pair of calls to **Direct3DDevice8.BeginScene** and **Direct3DDevice8.EndScene** must be made before any geometry is drawn.

See Also

Direct3DDevice8.GetViewport

Direct3DDevice8.ShowCursor

#Displays or hides the cursor.

object.**ShowCursor(_**
bShow **As Long) As Long**

IDH_Direct3DDevice8.SetViewport_graphicsvb
IDH_Direct3DDevice8.ShowCursor_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

bShow
If *bShow* is a non-zero value, the cursor is shown. If *bShow* is zero, the cursor is hidden.

Return Values

Value indicating whether the cursor was previously visible. This value is non-zero if the cursor was previously visible, or zero if the cursor was not previously visible.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method is independent of the Microsoft® Win32® API **ShowCursor** and **SetCursor** functions. Applications should either use the Win32 cursor or the Microsoft Direct3D® cursor, not both.

See Also

Direct3DDevice8.SetCursorPosition, **Direct3DDevice8.SetCursorProperties**

Direct3DDevice8.TestCooperativeLevel

#Reports the current cooperative-level status of the Microsoft® Direct3D® device for a windowed or full-screen application.

object.**TestCooperativeLevel() As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

IDH_Direct3DDevice8.TestCooperativeLevel_graphicsvb

Return Values

If the method succeeds, it returns D3D_OK, indicating that the device is operational and the calling application can continue executing.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values (see Remarks).

D3DERR_DEVICELOST

D3DERR_DEVICENOTRESET

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If the device is lost but cannot be restored at the current time, **Direct3DDevice8.TestCooperativeLevel** returns the D3DERR_DEVICELOST return code. This would be the case, for example, when a full-screen device has lost focus. If an application detects a lost device, it should pause and periodically call **TestCooperativeLevel** until it receives a return value of D3DERR_DEVICENOTRESET. The application may then attempt to reset the device by calling **Direct3DDevice8.Reset** and, if this succeeds, restore the necessary resources and resume normal operation. Note that **Direct3DDevice8.Present** will return D3DERR_DEVICELOST if the device is either "lost" or "not reset".

Direct3DDevice8.UpdateTexture

[#]Updates the dirty portions of a texture.

object.**UpdateTexture(** _
SourceTexture **As Direct3DBaseTexture8**, _
DestinationTexture **As Direct3DBaseTexture8)**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

SourceTexture
A **Direct3DBaseTexture8** object, representing the source texture. The source texture must be in system memory (D3DPOOL_SYSTEMMEM).

DestinationTexture

IDH_Direct3DDevice8.UpdateTexture_graphicsvb

A **Direct3DBaseTexture8** object, representing the destination texture. The destination texture must be in the D3DPOOL_DEFAULT memory pool.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

You can dirty a portion of a texture by locking it, or by calling one of the following methods.

- **Direct3DCubeTexture8.AddDirtyRect**
- **Direct3DTexture8.AddDirtyRect**
- **Direct3DVolumeTexture8.AddDirtyBox**
- **Direct3DDevice8.CopyRects**

UpdateTexture retrieves the dirty portions of the texture by calculating what has been accumulated since the last update operation.

This method fails if the textures are of different types, if their bottom-level buffers are of different sizes, and also if their matching levels do not match. For example, consider a six-level source texture with the following dimensions.

```
32×16, 16×8, 8×4, 4×2, 2×1, 1×1
```

This six-level source texture could be the source for the following one-level destination.

```
1×1
```

For the following two-level destination.

```
2×1, 1×1
```

Or, for the following three-level destination.

```
4×2, 2×1, 1×1
```

In addition, this method will fail if the textures are of different formats. If the destination texture has fewer levels than the source, only the matching levels are copied.

If the source texture has dirty regions, the copy may be optimized by restricting the copy to only those regions. It is not guaranteed that only those bytes marked dirty will be copied.

Direct3DDevice8.ValidateDevice

#Reports the device's ability to render the current texture-blending operations and arguments in a single pass.

object.**ValidateDevice() As Long**

Parts

object
Object expression that resolves to a **Direct3DDevice8** object.

Return Values

If the method succeeds, the return value is the number of rendering passes to complete the desired effect through multipass rendering.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_CONFLICTINGTEXTUREFILTER

D3DERR_CONFLICTINGTEXTUREPALETTE

D3DERR_TOOMANYOPERATIONS

D3DERR_UNSUPPORTEDALPHAARG

D3DERR_UNSUPPORTEDALPHAOPERATION

D3DERR_UNSUPPORTEDCOLORARG

D3DERR_UNSUPPORTEDCOLOROPERATION

D3DERR_UNSUPPORTEDFACTORVALUE

D3DERR_UNSUPPORTEDTEXTUREFILTER

D3DERR_WRONGTEXTUREFORMAT

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **ValidateDevice** method should be used only to validate scenarios when other capabilities are deficient. For example, in a multistage texturing scenario, you could

---

IDH_Direct3DDevice8.ValidateDevice_graphicsvb

query the **MaxTextureBlendStages** and **MaxSimultaneousTextures** members of a **D3DCAPS8** type to determine if multistage texturing is possible on the device.

Current hardware does not necessarily implement all possible combinations of operations and arguments. You can determine whether a particular blending operation can be performed with given arguments by setting the desired blending operation, and then calling the **ValidateDevice** method.

The **ValidateDevice** method uses the current render states, textures, and texture-stage states to perform validation at the time of the call. Changes to these factors after the call invalidate the previous result, and the method must be called again before rendering a scene.

Using diffuse iterated values, either as an argument or as an operation (D3DTA_DIFFUSE or D3DTOP_BLENDDIFFUSEALPHA), is rarely supported on current hardware. Most hardware can introduce iterated color data only at the last texture operation stage.

Try to specify the texture (D3DTA_TEXTURE) for each stage as the first argument, rather than the second argument.

Many cards do not support use of diffuse or scalar values at arbitrary texture stages. Often, these are available only at the first or last texture-blending stage.

Many cards do not have a blending unit associated with the first texture that is capable of more than replicating alpha to color channels or inverting the input. Therefore, your application might need to use only the second texture stage, if possible. On such hardware, the first unit is presumed to be in its default state, which has the first color argument set to D3DTA_TEXTURE with the D3DTOP_SELECTARG1 operation.

Operations on the output alpha that are more intricate than, or substantially different from, the color operations are less likely to be supported.

Some hardware does not support simultaneous use of D3DTA_TFACTOR and D3DTA_DIFFUSE.

Many cards do not support simultaneous use of multiple textures and mipmapped trilinear filtering. If trilinear filtering has been requested for a texture involved in multitexture blending operations and validation fails, turn off trilinear filtering and revalidate. In this case, you might want to perform multipass rendering instead.

See Also

Direct3DDevice8.GetTextureStageState, **Direct3DDevice8.SetTextureStageState**

Direct3DIndexBuffer8

#Applications use the methods of the **Direct3DIndexBuffer8** class to manipulate an index buffer resource.

IDH_Direct3DIndexBuffer8_graphicsvb

The **Direct3DIndexBuffer8** class is obtained by calling the **Direct3DDevice8.CreateIndexBuffer** method.

The **Direct3DIndexBuffer8** class implements the following **Direct3DResource8** methods, which can be organized into these groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The methods of the **Direct3DIndexBuffer8** class can be organized into the following groups.

Information	**GetDesc**
Locking	**Lock**
	Unlock

See Also

Direct3DDevice8.CreateIndexBuffer

Direct3DIndexBuffer8.GetDesc

#Retrieves a description of the index buffer resource.

object.**GetDesc(** _
Desc **As D3DINDEXBUFFER_DESC)**

Parts

object
Object expression that resolves to a **Direct3DIndexBuffer8** object.

Desc
A **D3DINDEXBUFFER_DESC** type, describing the returned index buffer.

IDH_Direct3DIndexBuffer8.GetDesc_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if the parameter is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DIndexBuffer8.Lock

#Locks a range of index data and obtains access to the index buffer memory.

object.**Lock(** _
OffsetToLock **As Long**, _
SizeToLock **As Long**, _
Data **As Long**, _
Flags **As Long)**

Parts

object
Object expression that resolves to a **Direct3DIndexBuffer8** object.

OffsetToLock
Offset into the index data to lock, in bytes.

SizeToLock
Size of the index data to lock, in bytes.

Data
First element of an array of **Long** values, filled with the returned index data

Flags
Combination of zero or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing how the index buffer memory should be locked.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

To lock, read, and unlock index data with a single function call, use **D3DIndexBuffer8GetData**.

IDH_Direct3DIndexBuffer8.Lock_graphicsvb

To lock fill, and unlock index data with a single function call, use **D3DIndexBuffer8SetData**.

When working with index buffers, you can make multiple lock calls; however, you must ensure that the number of lock calls match the number of unlock calls. DrawPrimitive calls will not succeed with any outstanding lock count on any currently set index buffer.

See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

See Also

Direct3DIndexBuffer8.Unlock, **D3DIndexBuffer8GetData**, **D3DIndexBuffer8SetData**

Direct3DIndexBuffer8.Unlock

#Unlocks index data.

object.**Unlock()**

Parts

object
Object expression that resolves to a **Direct3DIndexBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

To lock, read, and unlock index data with a single function call, use **D3DIndexBuffer8GetData**.

To lock fill, and unlock index data with a single function call, use **D3DIndexBuffer8SetData**.

IDH_Direct3DIndexBuffer8.Unlock_graphicsvb

See Also

Direct3DIndexBuffer8.Lock, **D3DIndexBuffer8SetData**, **D3DIndexBuffer8SetData**

Direct3DResource8

#Applications use the methods of the **Direct3DResource8** class to query and prepare resources.

To create a texture resource, you can call one of the following methods.

- **Direct3DDevice8.CreateCubeTexture**
- **Direct3DDevice8.CreateTexture**
- **Direct3DDevice8.CreateVolumeTexture**

To create a geometry-oriented resource, you can call one of the following methods.

- **Direct3DDevice8.CreateIndexBuffer**
- **Direct3DDevice8.CreateVertexBuffer**

The methods of the **Direct3DResource8** class can be organized into the following groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

Direct3DResource8.FreePrivateData

#Frees the specified private data associated with this resource.

object.**FreePrivateData(** _
RefGuid **As DXGUID)**

IDH_Direct3DResource8_graphicsvb
IDH_Direct3DResource8.FreePrivateData_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DResource8** object.

RefGuid
A **DXGUID** type, the globally unique identifier that identifies the private data to free.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DResource8**.

- **Direct3DBaseTexture8**
- **Direct3DCubeTexture8**
- **Direct3DIndexBuffer8**
- **Direct3DTexture8**
- **Direct3DVertexBuffer8**
- **Direct3DVolumeTexture8**

See Also

Direct3DResource8.GetPrivateData, **Direct3DResource8.SetPrivateData**

Direct3DResource8.GetDevice

#Retrieves the device associated with a resource.

object.**GetDevice(_**
Riid **As GUID) As Direct3DDevice8**

IDH_Direct3DResource8.GetDevice_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DResource8** object.

Riid
A **GUID** type, the reference identifier of the device being requested.

Return Values

An **Unknown** object to fill with the device object, if the query succeeds.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DResource8**.

- **Direct3DBaseTexture8**
- **Direct3DCubeTexture8**
- **Direct3DIndexBuffer8**
- **Direct3DTexture8**
- **Direct3DVertexBuffer8**
- **Direct3DVolumeTexture8**

Direct3DResource8.GetPriority

#Retrieves the priority for this resource.

object.**GetPriority() As Long**

Parts

object
Object expression that resolves to a **Direct3DResource8** object.

IDH_Direct3DResource8.GetPriority_graphicsvb

Return Values

Returns a **Long** value, indicating the priority of the resource.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DResource8**.

- **Direct3DBaseTexture8**
- **Direct3DCubeTexture8**
- **Direct3DIndexBuffer8**
- **Direct3DTexture8**
- **Direct3DVertexBuffer8**
- **Direct3DVolumeTexture8**

Remarks

GetPriority is used for priority control of managed resources. This method returns 0 on non-managed resources.

Priorities are used to determine when managed resources are to be removed from memory. A resource assigned a low priority is removed before a resource with a high priority. If two resources have the same priority, the resource that was used more recently is kept in memory; the other resource is removed. Managed resources have a default priority of 0.

See Also

Direct3DResource8.SetPriority

Direct3DResource8.GetPrivateData

#Copies the private data associated with the resource to a provided buffer.

```
object.GetPrivateData( _
   RefGuid As DXGUID, _
   Data As Any, _
```

IDH_Direct3DResource8.GetPrivateData_graphicsvb

SizeOfData **As Long)**

Parts

object
Object expression that resolves to a **Direct3DResource8** object.

RefGuid
A **DXGUID** type, the globally unique identifier that identifies the private data to retrieve.

Data
A previously allocated buffer to fill with the requested private data if the call succeeds. The application calling this method is responsible for allocating and releasing this buffer.

SizeOfData
Size of the buffer at *Data*, in bytes. If this value is less than the actual size of the private data (such as 0), the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_MOREDATA

D3DERR_NOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DResource8**.

- **Direct3DBaseTexture8**
- **Direct3DCubeTexture8**
- **Direct3DIndexBuffer8**
- **Direct3DTexture8**
- **Direct3DVertexBuffer8**
- **Direct3DVolumeTexture8**

See Also

Direct3DResource8.FreePrivateData, **Direct3DResource8.SetPrivateData**

Direct3DResource8.GetType

#Returns the type of the resource.

object.**GetType() As CONST_D3DRESOURCETYPE**

Parts

object
Object expression that resolves to a **Direct3DResource8** object.

Return Values

Returns a member of the **CONST_D3DRESOURCETYPE** enumeration, identifying the type of the resource.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DResource8**.

- **Direct3DBaseTexture8**
- **Direct3DCubeTexture8**
- **Direct3DIndexBuffer8**
- **Direct3DTexture8**
- **Direct3DVertexBuffer8**
- **Direct3DVolumeTexture8**

Direct3DResource8.PreLoad

#Preloads a managed resource.

IDH_Direct3DResource8.GetType_graphicsvb
IDH_Direct3DResource8.PreLoad_graphicsvb

object.**PreLoad()**

Parts

object
Object expression that resolves to a **Direct3DResource8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DResource8**.

- **Direct3DBaseTexture8**
- **Direct3DCubeTexture8**
- **Direct3DIndexBuffer8**
- **Direct3DTexture8**
- **Direct3DVertexBuffer8**
- **Direct3DVolumeTexture8**

Remarks

Calling this method indicates that the application will need this managed resource shortly. This method has no effect on nonmanaged resources.

PreLoad detects "thrashing" conditions where more resources are being used in each frame than can fit in video memory simultaneously. Under such circumstances **Preload** silently does nothing.

Direct3DResource8.SetPriority

#Assigns the resource-management priority for this resource.

object.**SetPriority(_**
PriorityNew **As Long) As Long**

IDH_Direct3DResource8.SetPriority_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DResource8** object.

PriorityNew
Long value that specifies the new resource-management priority for the resource.

Return Values

Returns the previous priority value for the resource.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **Direct3DResource8**.

- **Direct3DBaseTexture8**
- **Direct3DCubeTexture8**
- **Direct3DIndexBuffer8**
- **Direct3DTexture8**
- **Direct3DVertexBuffer8**
- **Direct3DVolumeTexture8**

Remarks

SetPriority is used for priority control of managed resources. This method returns 0 on nonmanaged resources.

Priorities are used to determine when managed resources are to be removed from memory. A resource assigned a low priority is removed before a resource with a high priority. If two resources have the same priority, the resource that was used more recently is kept in memory; the other resource is removed. Managed resources have a default priority of 0.

See Also

Direct3DResource8.GetPriority

Direct3DResource8.SetPrivateData

[#]Associates data with the resource that is intended for use by the application, not by Microsoft® Direct3D®. Data is passed by value, and multiple sets of data can be associated with a single resource.

object.**SetPrivateData(** _
RefGuid **As DXGUID**, _
Data **As Any**, _
SizeOfData **As Long**, _
Flags **As Long)**

Parts

object
Object expression that resolves to a **Direct3DResource8** object.

RefGuid
A **DXGUID** type, the globally unique identifier that identifies the private data to set.

Data
A buffer that contains the data to be associated with the resource.

SizeOfData
Size of the buffer at *Data*, in bytes.

Flags
Value that describes the type of data being passed, or indicates to the application that the data should be invalidated when the resource changes. This parameter can be the value defined by the **CONST_D3DSPDFLAGS** enumeration, or 0. If no flags are specified, Direct3D allocates memory to hold the data within the buffer and copies the data into the new buffer. The buffer allocated by Direct3D is automatically freed, as appropriate.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

[#] IDH_Direct3DResource8.SetPrivateData_graphicsvb

Applies To

This method applies to the following classes, which implement methods from **Direct3DResource8**.

- **Direct3DBaseTexture8**
- **Direct3DCubeTexture8**
- **Direct3DIndexBuffer8**
- **Direct3DTexture8**
- **Direct3DVertexBuffer8**
- **Direct3DVolumeTexture8**

Remarks

Direct3D does not manage the memory at *Data*. If this buffer was dynamically allocated, it is the caller's responsibility to free the memory.

See Also

Direct3DResource8.FreePrivateData, **Direct3DResource8.GetPrivateData**

Direct3DSurface8

#Applications use the methods of the **Direct3DSurface8** class to query and prepare surfaces.

The methods of the **Direct3DSurface8** class can be organized into the following groups.

Devices	**GetDevice**
Information	**GetContainer**
	GetDesc
Locking Surfaces	**LockRect**
	UnlockRect
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData

See Also

Surface Interfaces

IDH_Direct3DSurface8_graphicsvb

Direct3DSurface8.FreePrivateData

#Frees the specified private data associated with this surface.

object.**FreePrivateData(_**
RefGuid **As DXGUID)**

Parts

object
Object expression that resolves to a **Direct3DSurface8** object.

RefGuid
A **DXGUID** type, the globally unique identifier that identifies the private data to free.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DSurface8.GetPrivateData, **Direct3DSurface8.SetPrivateData**

Direct3DSurface8.GetContainer

#Provides access to the parent cube texture or texture (mipmap) object, if this surface is a child level of a cube texture or a mipmap.

object.**GetContainer(_**
Riid **As GUID) As Unknown**

Parts

object
Object expression that resolves to a **Direct3DSurface8** object.

Riid
Reference identifier of the container being requested.

IDH_Direct3DSurface8.FreePrivateData_graphicsvb
IDH_Direct3DSurface8.GetContainer_graphicsvb

Return Values

An **Unknown** object to fill with the container object, if the query succeeds. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If the surface is created using **Direct3DDevice8.CreateImageSurface**, **Direct3Ddevice8.CreateRenderTarget**, or **Direct3DDevice8.CreateDepthStencilSurface**, the surface is considered stand alone. In this case, **GetContainer** will return the Direct3D device used to create the surface.

Direct3DSurface8.GetDesc

#Retrieves a description of the surface.

object.**GetDesc(** _
Desc **As D3DSURFACE_DESC)**

Parts

object
Object expression that resolves to a **Direct3DSurface8** object.

Desc
A **D3DSURFACE_DESC** type, describing the surface.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DSurface8.GetDesc_graphicsvb

Direct3DSurface8.GetDevice

#Retrieves the device associated with a surface.

object.**GetDevice() As Direct3DDevice8**

Parts

object
Object expression that resolves to a **Direct3DSurface8** object.

Return Values

The **Direct3DDevice8** associated with the surface, if the query succeeds.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method enables navigation to the owning device object.

Direct3DSurface8.GetPrivateData

#Copies the private data associated with the surface to a provided buffer.

object.**GetPrivateData(_**
RefGuid **As DXGUID, _**
Data **As Any, _**
SizeOfData **As Long)**

Parts

object
Object expression that resolves to a **Direct3DSurface8** object.

RefGuid
A **DXGUID** type, the globally unique identifier that identifies the private data to retrieve.

Data

IDH_Direct3DSurface8.GetDevice_graphicsvb
IDH_Direct3DSurface8.GetPrivateData_graphicsvb

A previously allocated buffer to fill with the requested private data if the call succeeds. The application calling this method is responsible for allocating and releasing this buffer.

SizeOfData

Size of the buffer at *Data*, in bytes. If this value is less than the actual size of the private data (such as 0), the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_MOREDATA

D3DERR_NOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DSurface8.FreePrivateData, **Direct3DSurface8.SetPrivateData**

Direct3DSurface8.LockRect

#Locks a rectangle on a surface.

object.**LockRect(** _
LockedRect **As D3DLOCKED_RECT**, _
RECT **As Any**, _
Flags **As Long)**

Parts

object

Object expression that resolves to a **Direct3DSurface8** object.

LockedRect

A **D3DLOCKED_RECT** type, describing the locked region.

RECT

Rectangle to lock. Specified by a **RECT** type, or ByVal 0 to expand the dirty region to cover the entire surface.

Flags

IDH_Direct3DSurface8.LockRect_graphicsvb

A combination of zero or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing the type of lock to perform.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

A multisample backbuffer cannot be locked.

See Also

Direct3DSurface8.UnlockRect

Direct3DSurface8.SetPrivateData

#Associates data with the surface that is intended for use by the application, not by Microsoft® Direct3D®.

object.**SetPrivateData(** _
RefGuid **As DXGUID**, _
Data **As Any**, _
SizeOfData **As Long**, _
Flags **As Long)**

Parts

object
Object expression that resolves to a **Direct3DSurface8** object.

RefGuid
A **DXGUID** type, the globally unique identifier that identifies the private data to set.

Data
A buffer that contains the data to associate with the surface.

SizeOfData
Size of the buffer at *Data*, in bytes.

Flags
Value that describes the type of data being passed, or indicates to the application that the data should be invalidated when the resource changes. This parameter

IDH_Direct3DSurface8.SetPrivateData_graphicsvb

can be the value defined by the **CONST_D3DSPDFLAGS** enumeration, or 0. If no flags are specified, Direct3D allocates memory to hold the data within the buffer and copies the data into the new buffer. The buffer allocated by Direct3D is automatically freed, as appropriate.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Direct3D does not manage the memory at *Data*. If this buffer was dynamically allocated, it is the caller's responsibility to free the memory.

Data is passed by value, and multiple sets of data can be associated with a single surface.

See Also

Direct3DSurface8.FreePrivateData, **Direct3DSurface8.GetPrivateData**

Direct3DSurface8.UnlockRect

#Unlocks a rectangle on a surface.

object.**UnlockRect()**

Parts

object
Object expression that resolves to a **Direct3DSurface8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DSurface8.UnlockRect_graphicsvb

See Also

Direct3DSurface8.LockRect

Direct3DSwapChain8

#Applications use the methods of the **Direct3DSwapChain8** class to manipulate a swap chain.

There is always at least one swap chain, known as the implicit swap chain, for each device. However, an additional swap chain for rendering multiple views from the same device can be created by calling the **Direct3DDevice8.CreateAdditionalSwapChain** method.

The methods of the **Direct3DSwapChain8** class can be organized into the following groups.

Presentation	**Present**
Surface Management	**GetBackBuffer**

See Also

Direct3DDevice8.CreateAdditionalSwapChain

Direct3DSwapChain8.GetBackBuffer

#Retrieves a back buffers from the swap chain of the device.

object.**GetBackBuffer(_**
BackBuffer **As Long**, **_**
BufferType **As CONST_D3DBACKBUFFERTYPE) As Direct3DSurface8**

Parts

object
Object expression that resolves to a **Direct3DSwapChain8** object.

BackBuffer
Index of the back buffer object to return. Back buffers are numbered from 0 to the total number of back buffers - 1. A value of 0 returns the first back buffer, not the front buffer. The front buffer is not accessible through this method.

BufferType
Stereo view is not supported in DirectX 8.0, so the only valid value for this parameter is D3DBACKBUFFER_TYPE_MONO.

IDH_Direct3DSwapChain8_graphicsvb
IDH_Direct3DSwapChain8.GetBackBuffer_graphicsvb

Return Values

A **Direct3DSurface8** object, representing the returned back buffer surface.

Error Codes

If *BackBuffer* exceeds or equals the total number of back buffers, an error is raised and **Err.Number** may be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DSwapChain8.Present

#Presents the contents of the next in the sequence of back buffers owned by the swap chain.

object.**Present(** _
SourceRect **As Any**, _
DestRect **As Any**, _
DestWindowOverride **As Long**, _
DirtyRegion **As Any)**

Parts

object
Object expression that resolves to a **Direct3DSwapChain8** object.

SourceRect
Value that must be ByVal 0 unless the swap chain was created with D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC. *SourceRect* is a type containing the source rectangle. If ByVal 0, the entire source surface is presented. If the rectangle exceeds the source surface, the rectangle is clipped to the source surface.

DestRect
Value that must be ByVal 0 unless the swap chain was created with D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC. *DestRect* is a type containing the destination rectangle, in window client coordinates. If ByVal 0, the entire client area is filled. If the rectangle exceeds the destination client area, the rectangle is clipped to the destination client area.

DestWindowOverride
Destination window whose client area is taken as the target for this presentation.

DirtyRegion
This parameter is not used and should be set to ByVal 0.

IDH_Direct3DSwapChain8.Present_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If necessary, a stretch operation is applied to transfer the pixels within the source rectangle to the destination rectangle in the client area of the target window.

The following code fragment shows how to call the default case for **Present**.

```
Dim swapChain As Direct3DSwapChain8
swapChain.Present ByVal 0, ByVal 0, 0, ByVal 0
```

For the default case, note that you must specify ByVal 0 for the parameters that expect type **Any**. If you specify 0 (zero) for a parameter of type **Any**, an error will be generated and the method will fail.

See Also

Direct3DDevice8.Reset

Direct3DTexture8

#Applications use the methods of the **Direct3DTexture8** class to manipulate a texture resource.

The **Direct3DTexture8** class is obtained by calling the **Direct3DDevice8.CreateTexture** method.

The **Direct3DTexture8** class implements the following **Direct3DResource8** methods, which can be organized into these groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData

IDH_Direct3DTexture8_graphicsvb

	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The **Direct3DTexture8** class implements the following **Direct3DBaseTexture8** methods, which can be organized into these groups.

Detail	**GetLOD**
	SetLOD
Information	**GetLevelCount**

The methods of the **Direct3DTexture8** class can be organized into the following groups.

Information	**GetLevelDesc**
Locking Surfaces	**LockRect**
	UnlockRect
Miscellaneous	**AddDirtyRect**
	GetSurfaceLevel

See Also

Direct3DDevice8.CreateTexture

Direct3DTexture8.AddDirtyRect

#Adds a dirty region to a texture resource.

object.**AddDirtyRect(** _
DirtyRect **As Any)**

Parts

object
Object expression that resolves to a **Direct3DTexture8** object.

DirtyRect
A **RECT** type, specifying the dirty region to add. Specifying ByVal 0 expands the dirty region to cover the entire texture.

IDH_Direct3DTexture8.AddDirtyRect_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Applications can optimize what subset of a resource is copied by specifying dirty regions on the resource.

Direct3DTexture8.GetLevelDesc

#Retrieves a level description of a texture resource.

object.**GetLevelDesc(_**
Level **As Long, _**
Desc **As D3DSURFACE_DESC)**

Parts

object
Object expression that resolves to a **Direct3DTexture8** object.

Level
Identifies a level of the texture resource. This method returns a surface description for the level specified by this parameter.

Desc
A **D3DSURFACE_DESC** type, describing the returned mipmap.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if one or more of the parameters are invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DTexture8.GetSurfaceLevel

#Retrieves the specified texture surface level.

object.**GetSurfaceLevel(_**
Level **As Long) As Direct3DSurface8**

IDH_Direct3DTexture8.GetLevelDesc_graphicsvb
IDH_Direct3DTexture8.GetSurfaceLevel_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DTexture8** object.

Level
Identifies a level of the texture resource. This method returns a surface for the level specified by this parameter. The top-level surface is denoted by 0.

Return Values

A **Direct3DSurface8** object, representing the returned surface.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DTexture8.LockRect

#Locks a rectangle on a texture resource.

```
object.LockRect( _
    Level As Long, _
    LockedRect As D3DLOCKED_RECT, _
    RECT As Any, _
    Flags As Long)
```

Parts

object
Object expression that resolves to a **Direct3DTexture8** object.

Level
Specifies the level of the texture resource to lock.

LockedRect
A **D3DLOCKED_RECT** type, describing the locked region.

RECT
Rectangle to lock. Specified by a **RECT** type, or ByVal 0 to expand the dirty region to cover the entire texture.

Flags

IDH_Direct3DTexture8.LockRect_graphicsvb

A combination of zero or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing the type of lock to perform.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Textures created in D3DPOOL_DEFAULT are not-lockable.

The only lockable format for a depth-stencil surface is D3DFMT_D16_LOCKABLE.

A multisample backbuffer cannot be locked. Video memory textures cannot be locked, but they must be modified by calling **Direct3Ddevice8.CopyRects** or **Direct3Ddevice8.UpdateTexture**. There are exceptions for some proprietary driver pixel formats that Microsoft® DirectX® 8.0 does not recognize. These can be locked.

See Also

Direct3DTexture8.UnlockRect

Direct3DTexture8.UnlockRect

#Unlocks a rectangle on a texture resource.

object.**UnlockRect(** _
Level **As Long)**

Parts

object
Object expression that resolves to a **Direct3DTexture8** object.

Level
Specifies the level of the texture resource to unlock.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

IDH_Direct3DTexture8.UnlockRect_graphicsvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DTexture8.LockRect

Direct3DVertexBuffer8

#Applications use the methods of the **Direct3DVertexBuffer8** class to manipulate an index buffer resource.

The **Direct3DVertexBuffer8** class is obtained by calling the **Direct3DDevice8.CreateVertexBuffer** method.

The **Direct3DVertexBuffer8** class implements the following **Direct3DResource8** methods, which can be organized into these groups.

Devices	**GetDevice**
Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The methods of the **Direct3DVertexBuffer8** class can be organized into the following groups.

Information	**GetDesc**
Locking	**Lock**
	Unlock

See Also

Direct3DDevice8.CreateVertexBuffer

Direct3DVertexBuffer8.GetDesc

#Retrieves a description of the vertex buffer resource.

IDH_Direct3DVertexBuffer8_graphicsvb
IDH_Direct3DVertexBuffer8.GetDesc_graphicsvb

object.**GetDesc(** _
Desc **As D3DVERTEXBUFFER_DESC)**

Parts

object
Object expression that resolves to a **Direct3DVertexBuffer8** object.

Desc
A **D3DVERTEXBUFFER_DESC** type, describing the returned vertex buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if the parameter is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DVertexBuffer8.Lock

#Locks a range of vertex data and obtains access to the vertex buffer memory.

object.**Lock(** _
OffsetToLock **As Long**, _
SizeToLock **As Long**, _
Data **As Long**, _
Flags **As Long)**

Parts

object
Object expression that resolves to a **Direct3DVertexBuffer8** object.

OffsetToLock
Offset into the vertex data to lock, in bytes.

SizeToLock
Size of the vertex data to lock, in bytes. Specify 0 to lock the entire vertex buffer.

Data
First element of an array of **Long** values, filled with the returned vertex data.

Flags
Combination of zero or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing how the vertex buffer memory should be locked.

IDH_Direct3DVertexBuffer8.Lock_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

To lock, read, and unlock vertex data with a single function call, use **D3DVertexBuffer8GetData**.

To lock fill, and unlock vertex data with a single function call, use **D3DVertexBuffer8SetData**.

When working with vertex buffers, you can make multiple lock calls; however, you must ensure that the number of lock calls match the number of unlock calls. DrawPrimitive calls will not succeed with any outstanding lock count on any currently set vertex buffer.

See Using Dynamic Vertex and Index Buffers for information on using D3DLOCK_DISCARD or D3DLOCK_NOOVERWRITE for the *Flags* parameter of the **Lock** method.

See Also

Direct3DVertexBuffer8.Unlock, **D3DVertexBuffer8GetData**, **D3DVertexBuffer8SetData**

Direct3DVertexBuffer8.Unlock

#Unlocks vertex data.

object.**Unlock()**

Parts

object
Object expression that resolves to a **Direct3DVertexBuffer8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DVertexBuffer8.Unlock_graphicsvb

Remarks

To lock, read, and unlock vertex data with a single function call, use **D3DVertexBuffer8GetData**.

To lock fill, and unlock vertex data with a single function call, use **D3DVertexBuffer8SetData**.

See Also

Direct3DVertexBuffer8.Lock, **D3DVertexBuffer8GetData**, **D3DVertexBuffer8SetData**

Direct3DVolume8

#Applications use the methods of the **Direct3DVolume8** class to manipulate volume resources.

The **Direct3DVolume8** class is obtained by calling the **Direct3DVolumeTexture8.GetVolumeLevel** method.

The methods of the **Direct3DVolume8** class can be organized into the following groups.

Devices	**GetDevice**
Information	**GetContainer**
	GetDesc
Locking Volumes	**LockBox**
	UnlockBox
Private Volume Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData

Direct3DVolume8.FreePrivateData

#Frees the specified private data associated with this volume.

```
object.FreePrivateData( _
    RefGuid As DXGUID)
```

IDH_Direct3DVolume8_graphicsvb
IDH_Direct3DVolume8.FreePrivateData_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DVolume8** object.

RefGuid
A **DXGUID** type, the globally unique identifier that identifies the private data to free.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DVolume8.GetPrivateData, **Direct3DVolume8.SetPrivateData**

Direct3DVolume8.GetContainer

#Provides access to the parent volume texture object, if this surface is a child level of a volume texture.

object.**GetContainer(_**
Riid **As GUID) As Unknown**

Parts

object
Object expression that resolves to a **Direct3DVolume8** object.

Riid
Reference identifier of the volume being requested.

Return Values

An **Unknown** object to fill with the container object, if the query succeeds.

IDH_Direct3DVolume8.GetContainer_graphicsvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DVolume8.GetDevice

#Retrieves the device associated with a volume.

object.**GetDevice() As Direct3DDevice8**

Parts

object
Object expression that resolves to a **Direct3DVolume8** object.

Return Values

The **Direct3DDevice8** associated with the volume, if the query succeeds.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method enables navigation to the owning device object. For example, using **IID_IDirect3DDevice8** returns the Microsoft® Direct3D® interface for the object that created this buffer.

Direct3DVolume8.GetPrivateData

#Copies the private data associated with the volume to a provided buffer.

object.**GetPrivateData(** _
RefGuid **As DXGUID**, _
Data **As Any**, _
SizeOfData **As Long)**

IDH_Direct3DVolume8.GetDevice_graphicsvb
IDH_Direct3DVolume8.GetPrivateData_graphicsvb

Parts

object
Object expression that resolves to a **Direct3DVolume8** object.

RefGuid
A **DXGUID** type, the globally unique identifier that identifies the private data to retrieve.

Data
A previously allocated buffer to fill with the requested private data if the call succeeds. The application calling this method is responsible for allocating and releasing this buffer.

SizeOfData
Size of the buffer at *Data*, in bytes. If this value is less than the actual size of the private data (such as 0), the method sets this parameter to the required buffer size, and the method returns D3DERR_MOREDATA.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_MOREDATA

D3DERR_NOTFOUND

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DVolume8.FreePrivateData, **Direct3DVolume8.SetPrivateData**

Direct3DVolume8.GetDesc

#Retrieves a description of the volume.

```
object.GetDesc( _
   Desc As D3DVOLUME_DESC)
```

Parts

object
Object expression that resolves to a **Direct3DVolume8** object.

Desc

IDH_Direct3DVolume8.GetDesc_graphicsvb

A **D3DVOLUME_DESC** type, describing the volume.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if the parameter is invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DVolume8.LockBox

#Locks a box on a volume.

object.**LockBox(_**
LockedBox **As D3DLOCKED_BOX**, **_**
Box **As D3DBOX**, **_**
Flags **As Long)**

Parts

object
Object expression that resolves to a **Direct3DVolume8** object.

LockedBox
A **D3DLOCKED_BOX** type, describing the locked region.

Box
Box to lock. Specified by a **D3DBOX** type. Specifying ByVal 0 locks the entire volume.

Flags
A combination of zero or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing the type of lock to perform.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DVolume8.UnlockBox

IDH_Direct3DVolume8.LockBox_graphicsvb

Direct3DVolume8.SetPrivateData

[#]Associates data with the volume that is intended for use by the application, not by Microsoft® Direct3D®.

object.**SetPrivateData(** _
RefGuid **As DXGUID**, _
Data **As Any**, _
SizeOfData **As Long**, _
Flags **As Long)**

Parts

object
Object expression that resolves to a **Direct3DVolume8** object.

RefGuid
A **DXGUID** type, the globally unique identifier that identifies the private data to set.

Data
A buffer that contains the data to associate with the surface.

SizeOfData
Size of the buffer at *Data*, in bytes.

Flags
Value that describes the type of data being passed, or indicates to the application that the data should be invalidated when the resource changes. This parameter can be the value defined by the **CONST_D3DSPDFLAGS** enumeration, or 0. If no flags are specified, Direct3D allocates memory to hold the data within the buffer and copies the data into the new buffer. The buffer allocated by Direct3D is automatically freed, as appropriate.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

[#] IDH_Direct3DVolume8.SetPrivateData_graphicsvb

Remarks

Direct3D does not manage the memory at *Data*. If this buffer was dynamically allocated, it is the caller's responsibility to free the memory.

Data is passed by value, and multiple sets of data can be associated with a single volume.

See Also

Direct3DVolume8.FreePrivateData, **Direct3DVolume8.GetPrivateData**

Direct3DVolume8.UnlockBox

#Unlocks a box on a volume.

object.**UnlockBox()**

Parts

object
Object expression that resolves to a **Direct3DVolume8** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DVolume8.LockBox

Direct3DVolumeTexture8

#Applications use the methods of the **Direct3DVolumeTexture8** class to manipulate a volume texture resource.

The **Direct3DVolumeTexture8** class is obtained by calling the **Direct3DDevice8.CreateVolumeTexture** method.

The **Direct3DVolumeTexture8** class implements the following **Direct3DResource8** methods, which can be organized into these groups.

Devices	GetDevice

IDH_Direct3DVolume8.UnlockBox_graphicsvb
IDH_Direct3DVolumeTexture8_graphicsvb

Information	**GetType**
Private Surface Data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Resource Management	**GetPriority**
	PreLoad
	SetPriority

The **Direct3DVolumeTexture8** class implements the following **Direct3DBaseTexture8** methods, which can be organized into these groups.

Detail	**GetLOD**
	SetLOD
Information	**GetLevelCount**

The methods of the **Direct3DVolumeTexture8** class can be organized into the following groups.

Information	**GetLevelDesc**
Locking Volumes	**LockBox**
	UnlockBox
Miscellaneous	**AddDirtyBox**
	GetVolumeLevel

See Also

Direct3DDevice8.CreateVolumeTexture

Direct3DVolumeTexture8.AddDirtyBox

#Adds a dirty region to a volume texture resource.

object.**AddDirtyBox(_**
DirtyBox **As Any)**

Parts

object

IDH_Direct3DVolumeTexture8.AddDirtyBox_graphicsvb

Object expression that resolves to a **Direct3DVolumeTexture8** object.

DirtyBox
a **D3DBOX** type, specifying the dirty region to add. Specifying ByVal 0 locks the entire volume texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Applications can optimize what subset of a resource is copied by specifying boxes on the resource. However, the dirty regions may be expanded to optimize alignment.

Direct3DVolumeTexture8.GetLevelDesc

#Retrieves a level description of a volume texture resource.

object.**GetLevelDesc(_**
Level **As Long, _**
pDesc **As D3DVOLUME_DESC)**

Parts

object
Object expression that resolves to a **Direct3DVolumeTexture8** object.

Level
Identifies a level of the volume texture resource. This method returns a volume description for the level specified by this parameter.

pDesc
A **D3DVOLUME_DESC** type, describing the returned volume texture level.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL if one or more of the parameters are invalid.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DVolumeTexture8.GetLevelDesc_graphicsvb

Direct3DVolumeTexture8.GetVolumeLevel

#Retrieves the specified volume texture level.

object.**GetVolumeLevel(_**
Level As Long) As Direct3DVolume8

Parts

object
Object expression that resolves to a **Direct3DVolumeTexture8** object.

Level
Identifies a level of the volume texture resource. This method returns a volume for the level specified by this parameter.

Return Values

A **Direct3DVolume8** object, representing the returned volume level.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Direct3DVolumeTexture8.LockBox

#Locks a box on a volume texture resource.

object.**LockBox(_**
Level As Long, _
LockedVolume As D3DLOCKED_BOX, _
Box As Any, _
Flags As Long)

Parts

object
Object expression that resolves to a **Direct3DVolumeTexture8** object.

Level

IDH_Direct3DVolumeTexture8.GetVolumeLevel_graphicsvb
IDH_Direct3DVolumeTexture8.LockBox_graphicsvb

Specifies the level of the volume texture resource to lock.

LockedVolume
A **D3DLOCKED_BOX** type, describing the locked region.

Box
Volume to lock. Specified by a **D3DBOX** type. Specifying ByVal 0 locks the entire volume level.

Flags
A combination of zero or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing the type of lock to perform.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

Direct3DVolumeTexture8.UnlockBox

Direct3DVolumeTexture8.UnlockBox

#Locks a box on a volume texture resource.

object.**UnlockBox(** _
Level **As Long)**

Parts

object
Object expression that resolves to a **Direct3DVolumeTexture8** object.

Level
Specifies the level of the volume texture resource to lock.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_Direct3DVolumeTexture8.UnlockBox_graphicsvb

See Also

Direct3DVolumeTexture8.LockBox

Functions

This section contains reference information for the functions that you need to implement when you work with Microsoft® Direct3D®. The following functions are implemented.

- **D3DColorARGB**
- **D3DColorMake**
- **D3DColorRGBA**
- **D3DColorXRGB**
- **D3DIndexBuffer8GetData**
- **D3DVertexBuffer8GetData**
- **D3DIndexBuffer8SetData**
- **D3DVertexBuffer8SetData**

D3DColorARGB

#Initializes a color with the supplied alpha, red, green, and blue values.

```
D3DColorARGB( _
   a As Integer, _
   r As Integer, _
   g As Integer, _
   b As Integer) As Long
```

Parameters

a

Alpha value of the color. Values between 0 and 255.

r

Red value of the color. Values between 0 and 255.

g

Green value of the color. Values between 0 and 255.

b

Blue value of the color. Values between 0 and 255.

IDH_D3DColorARGB_graphicsvb

Return Values

Returns the **Long** value that corresponds to the supplied ARGB values.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Visual Basic Error Handling topic.

D3DColorMake

#Initializes a color with the supplied red, green, blue, and alpha floating-point values.

```
D3DColorMake( _
   r As Single, _
   g As Single, _
   b As Single, _
   a As Single) As Long
```

Parameters

r
Red value of the color. Values between 0.0 and 1.0.

g
Green value of the color. Values between 0.0 and 1.0.

b
Blue value of the color. Values between 0.0 and 1.0.

a
Blue value of the color. Values between 0.0 and 1.0.

Return Values

Returns the **Long** value that corresponds to the supplied RGBA values.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Visual Basic Error Handling topic.

IDH_D3DColorMake_graphicsvb

D3DColorRGBA

#Initializes a color with the supplied red, green, blue, and alpha values.

```
D3DColorRGBA( _
   a As Integer, _
   r As Integer, _
   g As Integer, _
   b As Integer) As Long
```

Parameters

r

Red value of the color. Values between 0 and 255.

g

Green value of the color. Values between 0 and 255.

b

Blue value of the color. Values between 0 and 255.

a

Alpha value of the color. Values between 0 and 255.

Return Values

Returns the **Long** value that corresponds to the supplied RGBA values.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Visual Basic Error Handling topic.

D3DColorXRGB

#Initializes a color with the supplied red, green, and blue values.

```
D3DColorXRGB( _
   r As Integer, _
   g As Integer, _
   b As Integer) As Long
```

Parameters

r

IDH_D3DColorRGBA_graphicsvb
IDH_D3DColorXRGB_graphicsvb

Red value of the color. Values between 0 and 255.

g
Green value of the color. Values between 0 and 255.

b
Blue value of the color. Values between 0 and 255.

Return Values

Returns the **Long** value that corresponds to the supplied RGB values.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Visual Basic Error Handling topic.

D3DIndexBuffer8GetData

#Retrieves data from an index buffer.

D3DIndexBuffer8GetData(_
IBuffer **As Direct3DIndexBuffer8, _**
Offset **As Long, _**
Size **As Long, _**
Flags **As Long, _**
Data **As Any) As Long**

Parameters

IBuffer
Direct3DIndexBuffer8 object, representing the index buffer from which to retrieve data.

Offset
Offset, in bytes, from the start of the buffer to where data retrieval begins.

Size
Size of the buffer, in bytes.

Flags
A combination of one or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing the type of lock to perform.

Data
Buffer to be filled with the retrieved data.

IDH_D3DIndexBuffer8GetData_graphicsvb

Return Values

If the method fails, D3DERR_INVALIDCALL can be returned.

Error Codes

Err.Number is not set by this function.

See Also

D3DIndexBuffer8SetData

D3DVertexBuffer8GetData

#Retrieves data from a vertex buffer.

D3DVertexBuffer8GetData(_
VBuffer **As Direct3DVertexBuffer8**, **_**
Offset **As Long**, **_**
Size **As Long**, **_**
Flags **As Long**, **_**
Data **As Any) As Long**

Parameters

VBuffer
Direct3DVertexBuffer8 object, representing the vertex buffer from which to retrieve data.

Offset
Offset, in bytes, from the start of the buffer to where data retrieval begins.

Size
Size of the buffer, in bytes.

Flags
A combination of one or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing the type of lock to perform.

Data
Buffer to be filled with the retrieved data.

Return Values

If the method fails, D3DERR_INVALIDCALL can be returned.

Error Codes

IDH_D3DVertexBuffer8GetData_graphicsvb

Err.Number is not set by this function.

See Also

D3DVertexBuffer8SetData

D3DIndexBuffer8SetData

#Sets data in an index buffer.

D3DIndexBuffer8SetData(_
IBuffer **As Direct3DIndexBuffer8**, _
Offset **As Long**, _
Size **As Long**, _
Flags **As Long**, _
Data **As Any) As Long**

Parameters

IBuffer
Direct3DIndexBuffer8 object, representing the index buffer into which to set data.

Offset
Offset, in bytes, from the start of the buffer to where data is set.

Size
Size of the buffer, in bytes.

Flags
A combination of one or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing the type of lock to perform.

Data
Buffer containing data to set.

Return Values

If the method fails, D3DERR_INVALIDCALL can be returned.

Error Codes

Err.Number is not set by this function.

See Also

D3DIndexBuffer8GetData

IDH_D3DIndexBuffer8SetData_graphicsvb

D3DVertexBuffer8SetData

#Sets data in a vertex buffer.

D3DVertexBuffer8SetData(_
VBuffer **As Direct3DVertexBuffer8**, _
Offset **As Long**, _
Size **As Long**, _
Flags **As Long**, _
Data **As Any) As Long**

Parameters

VBuffer
Direct3DVertexBuffer8 object, representing the vertex buffer into which to set data.

Offset
Offset, in bytes, from the start of the buffer to where data is set.

Size
Size of the buffer, in bytes.

Flags
A combination of one or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing the type of lock to perform.

Data
Buffer containing data to set.

Return Values

If the method fails, D3DERR_INVALIDCALL can be returned.

Error Codes

Err.Number is not set by this function.

See Also

D3DVertexBuffer8GetData

Vertex Shader Declarator Functions

This section covers the bit assembly functions used to compose the vertex shader declaration array.

- **D3DVSD_CONST**

IDH_D3DVertexBuffer8SetData_graphicsvb

- **D3DVSD_END**
- **D3DVSD_NOP**
- **D3DVSD_REG**
- **D3DVSD_SKIP**
- **D3DVSD_STREAM**
- **D3DVSD_STREAM_TESS**
- **D3DVSD_TESSNORMAL**
- **D3DVSD_TESSUV**

D3DVSD_CONST

Loads data into the vertex shader constant memory.

```
Function D3DVSD_CONST(ConstantAddress As Long, Count As Long) As
Long
    D3DVSD_CONST =
(D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_constMEM) Or _
    ((Count) * 2 ^ D3DVSD_CONSTCOUNTSHIFT) Or (ConstantAddress))
End Function
```

Parameters

ConstantAddress
Address of the constant array to begin filling data. Possible values range from 0 to 95.

Count
Number of constant vectors to load. See Remarks.

Remarks

D3DVSD_CONST and all the constants it uses are found in (*SDK Root*) \Samples\Multimedia\VBSamples\Common\D3DShaders.bas.

The following code example shows how this function might be used.

```
Dim Declarator(12) As Long

Declarator(0)  = D3DVSD_CONST(8, 1)
Declarator(1)  = FtoDW(1)
Declarator(2)  = FtoDW(1)
Declarator(3)  = FtoDW(1/255)
Declarator(4)  = FtoDW(1)
Declarator(5)  = D3DVSD_STREAM(0)
Declarator(6)  = D3DVSD_REG(0, D3DVSDT_FLOAT3)
```

```
Declarator(7)  = D3DVSD_REG(2, D3DVSDT_D3DCOLOR)
Declarator(8)  = D3DVSD_STREAM(1)
Declarator(9)  = D3DVSD_REG(1, D3DVSDT_FLOAT1)
Declarator(10) = D3DVSD_STREAM(2)
Declarator(11) = D3DVSD_REG(3, D3DVSDT_FLOAT2)
Declarator(12) = D3DVSD_END()
```

See Also

Direct3DDevice8.CreateVertexShader

D3DVSD_END

Generates an END token.

```
Function D3DVSD_END() As Long
      D3DVSD_END = &HFFFFFFFF
End Function
```

Parameters

None.

Remarks

D3DVSD_END and all the constants it uses are found in (*SDK Root*) \Samples\Multimedia\VBSamples\Common\D3DShaders.bas.

See Also

Direct3DDevice8.CreateVertexShader

D3DVSD_NOP

Generates an NOP token.

```
Function D3DVSD_NOP() As Long
   D3DVSD_NOP = 0
End Function
```

Parameters

None.

Remarks

D3DVSD_NOP and all the constants it uses are found in (*SDK Root*) \Samples\Multimedia\VBSamples\Common\D3DShaders.bas.

See Also

Direct3DDevice8.CreateVertexShader

D3DVSD_REG

Binds a single vertex register to a vertex element from the vertex stream.

```
Function D3DVSD_REG(VertexRegister As Long, dataType As Long) As Long
    D3DVSD_REG =
(D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_STREAMDATA) Or _
     ((dataType) * 2 ^ D3DVSD_DATATYPESHIFT) Or (VertexRegister))
End Function
```

Parameters

VertexRegister

Address of the vertex register. Possible values range from 0 through 15. For the fixed function pipeline, the input registers have the following fixed mapping.

D3DVSDE_POSITION
Register 0.

D3DVSDE_BLENDWEIGHT
Register 1.

D3DVSDE_NORMAL
Register 2.

D3DVSDE_PSIZE
Register 3.

D3DVSDE_DIFFUSE
Register 4.

D3DVSDE_SPECULAR
Register 5.

D3DVSDE_TEXCOORD0
Register 6.

D3DVSDE_TEXCOORD1
Register 7.

D3DVSDE_TEXCOORD2
Register 8.

D3DVSDE_TEXCOORD3
Register 9.

D3DVSDE_TEXCOORD4
Register 10.

D3DVSDE_TEXCOORD5
Register 11.

D3DVSDE_TEXCOORD6
Register 12.

D3DVSDE_TEXCOORD7
Register 13.

dataType
Specifies the dimensionality and arithmetic data type. The following values are defined.

D3DVSDT_D3DCOLOR
4-D packed unsigned bytes mapped to 0.0 to 1.0 range. In double word format this is ARGB, or in byte ordering it would be B, G, R, A.

D3DVSDT_FLOAT1
1-D float expanded to (*value*, 0.0, 0.0, 1.0)

D3DVSDT_FLOAT2
2-D float expanded to (*value*, *value*, 0.0, 1.0)

D3DVSDT_FLOAT3
3-D float expanded to (*value*, *value*, *value*, 1.0)

D3DVSDT_FLOAT4
4-D float.

D3DVSDT_UBYTE4
4-D unsigned byte. In double word format this is ABGR, or in byte ordering it would be R, G, B, A.

Remarks

D3DVSD_REG and all the constants it uses are found in (*SDK Root*) \Samples\Multimedia\VBSamples\Common\D3DShaders.bas.

See Also

Direct3DDevice8.CreateVertexShader

D3DVSD_SKIP

Specifies how many **Long**s to skip in the vertex.

```
Function D3DVSD_SKIP(LongCount As Long) As Long
   D3DVSD_SKIP =
(D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_STREAMDATA) Or _
            &H10000000 Or _
```

```
            ((LongCount) * 2 ^ D3DVSD_SKIPCOUNTSHIFT))
End Function
```

Parameters

LongCount
Number of **Long**s to skip in the vertex.

Remarks

D3DVSD_SKIP and all the constants it uses are found in (*SDK Root*) \Samples\Multimedia\VBSamples\Common\D3DShaders.bas.

See Also

Direct3DDevice8.CreateVertexShader

D3DVSD_STREAM

Sets the current stream.

```
Function D3DVSD_STREAM(StreamNumber As Long) As Long
   D3DVSD_STREAM =
(D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_STREAM) Or _
            (StreamNumber))
End Function
```

Parameters

StreamNumber
Specifies the stream from which to get data. Possible values range from 0 through MaxStreams - 1.

Remarks

D3DVSD_STREAM and all the constants it uses are found in (*SDK Root*) \Samples\Multimedia\VBSamples\Common\D3DShaders.bas.

See Also

Direct3DDevice8.CreateVertexShader

D3DVSD_STREAM_TESS

Sets the tessellator stream.

```
Function D3DVSD_STREAM_TESS() As Long
   D3DVSD_STREAM_TESS =
(D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_STREAM) Or _
                (D3DVSD_STREAMTESSMASK))
End Function
```

Parameters

None.

Remarks

D3DVSD_STREAM_TESS and all the constants it uses are found in (*SDK Root*) \Samples\Multimedia\VBSamples\Common\D3DShaders.bas.

See Also

Direct3DDevice8.CreateVertexShader

D3DVSD_TESSNORMAL

Enables tessellator-generated normals.

```
Function D3DVSD_TESSNORMAL(VertexRegisterIn As Long,
VertexRegisterOut As Long) As Long
   D3DVSD_TESSNORMAL =
(D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_TESSELLATOR) Or _
    ((VertexRegisterIn) * 2 ^ D3DVSD_VERTEXREGINSHIFT) Or _
    ((&H2&) * 2 ^ D3DVSD_DATATYPESHIFT) Or (VertexRegisterOut))
End Function
```

Parameters

VertexRegisterIn
Address of the vertex register whose input stream will be used in normal computation. Possible values range from 0 to 15.

VertexRegisterOut
Address of the vertex register to output the normal to. Possible values range from 0 to 15.

Remarks

D3DVSD_TESSNORMAL and all the constants it uses are found in (*SDK Root*) \Samples\Multimedia\VBSamples\Common\D3DShaders.bas.

See Also

Direct3DDevice8.CreateVertexShader

D3DVSD_TESSUV

Enables tessellator-generated surface parameters.

```
Function D3DVSD_TESSUV(VertexRegister As Long) As Long
   D3DVSD_TESSUV =
(D3DVSD_MAKETOKENTYPE(D3DVSD_TOKEN_TESSELLATOR) Or _
                &H10000000 Or _
                ((&H1&) * 2 ^ D3DVSD_DATATYPESHIFT) Or _
                (VertexRegister))
End Function
```

Parameters

VertexRegister
Address of the vertex register to output parameters. Possible values range from 0 to 15.

Remarks

D3DVSD_TESSUV and all the constants it uses are found in (*SDK Root*) \Samples\Multimedia\VBSamples\Common\D3DShaders.bas.

See Also

Direct3DDevice8.CreateVertexShader

Types

This section contains information about the types used with Microsoft® Direct3D®.

- **D3DADAPTER_IDENTIFIER8**
- **D3DBOX**

- **D3DCAPS8**
- **D3DCLIPSTATUS8**
- **D3DCOLORVALUE**
- **D3DDEVICE_CREATION_PARAMETERS**
- **D3DDISPLAYMODE**
- **D3DGAMMARAMP**
- **D3DINDEXBUFFER_DESC**
- **D3DLIGHT8**
- **D3DLINEPATTERN**
- **D3DLOCKED_BOX**
- **D3DLOCKED_RECT**
- **D3DLVERTEX**
- **D3DLVERTEX2**
- **D3DMATERIAL8**
- **D3DMATRIX**
- **D3DPLANE**
- **D3DPRESENT_PARAMETERS**
- **D3DQUATERNION**
- **D3DRANGE**
- **D3DRASTER_STATUS**
- **D3DRECT**
- **D3DRECTPATCH_INFO**
- **D3DSURFACE_DESC**
- **D3DTLVERTEX**
- **D3DTLVERTEX2**
- **D3DTRIPATCH_INFO**
- **D3DVECTOR**
- **D3DVECTOR2**
- **D3DVECTOR4**
- **D3DVERTEX**
- **D3DVERTEX2**
- **D3DVERTEXBUFFER_DESC**
- **D3DVIEWPORT8**
- **D3DVOLUME_DESC**

D3DADAPTER_IDENTIFIER8

#Contains information identifying the adapter.

```
Type D3DADAPTER_IDENTIFIER8
    Description(0 To 511) As Byte
    DeviceId As Long
    DeviceIdentifier As DXGUID
    Driver(0 To 511) As Byte
    DriverVersionHighPart As Long
    DriverVersionLowPart As Long
    Revision As Long
    SubSysId As Long
    VendorId As Long
    WHQLLevel As Long
End Type
```

Members

Description and **Driver**

For presentation to the user. These members should not be used to identify particular drivers because many different strings may be associated with the same device and driver from different vendors.

DeviceId

Identify the type of a particular chip set. The value may be zero if unknown.

DriverVersion, **DriverVersionLowPart**, and **DriverVersionHighPart**

Identify the version of the Microsoft® Direct3D® driver. It is legal to do < and > comparisons on the **Long** value. However, exercised caution if you use this element to identify problematic drivers. Instead, you should use **DeviceIdentifier**.

DeviceIdentifier

Track changes to the driver and chip set in order to generate a new profile for the graphics subsystem. This **DXGUID** is a unique identifier for the driver and chip set pair. You can also use **DeviceIdentifier** to identify particular problematic drivers.

Revision

Identify a particular chip set and its revision level. The value may be zero if unknown.

SubSysId

Identify a particular chip set and the subsystem. Typically this means the chip set's board. The value may be zero if unknown.

VendorId

IDH_D3DADAPTER_IDENTIFIER8_graphicsvb

Identify a particular chip set and its manufacturer. The value may be zero if unknown.

WHQLLevel

Determine the Windows Hardware Quality Lab (WHQL) certification level for this driver and device pair. The **Long** is a packed date structure defining the date of the release of the most recent WHQL test passed by the driver. It is legal to perform < and > operations on this value. The following illustrates the date format.

Bits

31-16: The year, a decimal number from 1999 upwards.

15-8: The month, a decimal number from 1 to 12.

7-0: The day, a decimal number from 1 to 31.

The following values are also used.

0

Not certified.

1

WHQL certified, but no date information is available.

Remarks

You can use the **VendorId**, **DeviceId**, **SubSysId**, and **Revision** members in tandem to identify particular chip sets. However, use these members with caution.

D3DBOX

#Defines a volume.

```
Type D3DBOX
    back As Long
    bottom As Long
    front As Long
    left As Long
    right As Long
    top As Long
End Type
```

Members

back, **bottom**, **front**, **left**, **right**, and **top**

The dimensions of the box.

IDH_D3DBOX_graphicsvb

Remarks

D3DBOX includes the left, top, and front edges; however, the right, bottom, and back edges are not included. For example, a box that is 100 units wide and begins at 0 (thus, including the points up to and including 99) would be expressed with a value of 0 for the **left** member and a value of 100 for the **right** member. Note that a value of 99 is not used for the **right** member.

The restrictions on side ordering observed for **D3DBOX** are left to right, top to bottom, and front to back.

D3DCAPS8

#Represents the capabilities of the hardware exposed through the Microsoft® Direct3D® object.

```
Type D3DCAPS8
    AdapterOrdinal As Long
    AlphaCmpCaps As Long
    caps As Long
    Caps2 As Long
    Caps3 As Long
    CubeTextureFilterCaps As Long
    CursorCaps As Long
    DestBlendCaps As Long
    DevCaps As Long
    DeviceType As Long
    ExtentsAdjust As Single
    FVFCaps As Long
    GuardBandBottom As Single
    GuardBandLeft As Single
    GuardBandRight As Single
    GuardBandTop As Single
    LineCaps As Long
    MaxActiveLights As Long
    MaxAnisotropy As Long
    MaxPixelShaderValue As Single
    MaxPointSize As Single
    MaxPrimitiveCount As Long
    MaxSimultaneousTextures As Long
    MaxStreams As Long
    MaxStreamStride As Long
    MaxTextureAspectRatio As Long
    MaxTextureBlendStages As Long
    MaxTextureHeight As Long
```

IDH_D3DCAPS8_graphicsvb

```
    MaxTextureRepeat As Long
    MaxTextureWidth As Long
    MaxUserClipPlanes As Long
    MaxVertexBlendMatrices As Long
    MaxVertexBlendMatrixIndex As Long
    MaxVertexIndex As Long
    MaxVertexShaderConst As Long
    MaxVertexW As Single
    MaxVolumeExtent As Long
    PixelShaderVersion As Long
    PresentationIntervals As Long
    PrimitiveMiscCaps As Long
    RasterCaps As Long
    ShadeCaps As Long
    SrcBlendCaps As Long
    StencilCaps As Long
    TextureAddressCaps As Long
    TextureCaps As Long
    TextureFilterCaps As Long
    TextureOpCaps As Long
    VertexProcessingCaps As Long
    VertexShaderVersion As Long
    VolumeTextureAddressCaps As Long
    VolumeTextureFilterCaps As Long
    ZCmpCaps As Long
End Type
```

Members

AdapterOrdinal

Adapter on which this Direct3DDevice object was created. This ordinal is valid only to pass to methods of the **Direct3D8** class that created this Direct3DDevice object. The **Direct3D8** class can always be retrieved by calling **Direct3DDevice8.GetDirect3D**.

AlphaCmpCaps

Alpha-test comparison capabilities. If this member contains only the D3DPCMPCAPS_ALWAYS capability or only the D3DPCMPCAPS_NEVER capability, the driver does not support alpha tests. Otherwise, the flags defined by the **CONST_D3DPCMPCAPSFLAGS** enumeration identify the individual comparisons that are supported for alpha testing.

caps

Driver-specific capability, defined by the **CONST_D3DCAPSFLAGS** enumeration.

Caps2

Driver-specific capabilities. Possible values are defined by the **CONST_D3DCAPS2FLAGS** enumeration.

Caps3
This value is not used.

CubeTextureFilterCaps
Texture-filtering capabilities for a Direct3DCubeTexture object. Per-stage filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple-texture blending with the **Direct3DDevice8** class. This member can be any combination of the general and per-stage texture-filtering flags defined by the **CONST_D3DPTFILTERCAPSFLAGS** enumeration.

CursorCaps
Bit mask indicating what hardware support is available for cursors. Possible values are defined by the **CONST_D3DCURSORCAPSFLAGS**.

DestBlendCaps
Destination-blending capabilities. This member can be one or more of the flags defined by the **CONST_D3DPBLENDCAPSFLAGS** enumeration.

DevCaps
Flags identifying the capabilities of the device. Possible values are defined by the **CONST_D3DDEVCAPSFLAGS** enumeration.

DeviceType
Member of the **CONST_D3DDEVTYPE** enumeration. Denotes the amount of emulated functionality for this device. The value of this parameter mirrors the value passed to the **Direct3D8.CreateDevice** call that created this device.

ExtentsAdjust
Number of pixels to adjust the extents rectangle outward to accommodate anti-aliasing kernels.

FVFCaps
Flexible vertex format capabilities. Possible values are defined by the **CONST_D3DFVFCAPSFLAGS** enumeration.

GuardBandBottom, **GuardBandLeft**, **GuardBandRight**, and **GuardBandTop**
Screen-space coordinates of the guard-band clipping region. Coordinates inside this rectangle but outside the viewport rectangle are automatically clipped.

LineCaps
Defines the capabilities for line-drawing primitives. Possible values are defined by the **CONST_D3DLINECAPS** enumeration.

MaxActiveLights
Maximum number of lights that can be active simultaneously. For a given physical device, this capability might vary across Direct3D Device Objects depending on the parameters supplied to **Direct3D8.CreateDevice**.

MaxAnisotropy
Maximum valid value for the D3DTSS_MAXANISOTROPY texture-stage state.

MaxPixelShaderValue

Maximum value of pixel shader arithmetic component. This value indicates the internal range of values supported for pixel color blending operations. Implementations must allow data within the range that they report to pass through pixel processing unmodified (unclampled). Normally, the value of this member is an absolute value. For example, a 1.0 indicates that the range is -1.0 to 1, and an 8.0 indicates that the range is -8.0 to 8.0. Note that the value 0.0 indicates that no signed range is supported; therefore, the range is 0 to 1.0 as in Microsoft DirectX® versions 6.0 and 7.0.

MaxPointSize
Maximum size of a point primitive. If set to 1.0f then device does not support point size control. The range is greater than or equal to 1.0.

MaxPrimitiveCount
Maximum number of primitives for each DrawPrimitive call.

MaxSimultaneousTextures
Maximum number of textures that can be simultaneously bound to the texture blending stages. This value indicates the number of textures that can be used in a single pass. In Microsoft® DirectX® 8.0, this indicates the number of texture registers supported by pixel shaders on this particular piece of hardware, and the number of texture declaration instructions that can be present.

MaxStreams
The maximum number of concurrent data streams for **Direct3DDevice8.SetStreamSource**. The valid range is 1 to 16. Note that if this value is 0, then the driver is not a Microsoft DirectX® 8.0 driver.

MaxStreamStride
Maximum stride for **Direct3DDevice8.SetStreamSource**.

MaxTextureAspectRatio
Maximum texture aspect ratio supported by the hardware; this is typically a power of 2.

MaxTextureBlendStages
Maximum number of texture-blending stages supported. This value is the number of blenders available. In the DirectX 8.0 programmable pixel pipeline, this should correspond to the number of instructions supported by pixel shaders on this particular implementation.

MaxTextureHeight and **MaxTextureWidth**
Maximum texture height and width for this device.

MaxTextureRepeat
Full range of the integer bits of the post-normalized texture indices. If the D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE bit is set, the device defers scaling by the texture size until after the texture address mode is applied. If not set, the device scales the texture indices by the texture size (largest level of detail) prior to interpolation.

If D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE is set, then the exact number of times a texture can be wrapped is **MaxTextureRepeat**. If D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE is not set, then the exact number of times a texture can be wrapped is (**MaxTextureRepeat** * texture

size). For example, the device has D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE set, **MaxTextureRepeat** is 32k, and the maximum texture size is 4k. The device uses 27 integer bits (plus 5 fraction bits in a 32 bit (signed) integer), and thus has enough to wrap a 4k texture 32k times (assuming the texture coordinates equally span the positive and negative texture coordinate range). Alternately, the device could not set D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE and set **MaxTextureRepeat** to 2**27.

MaxUserClipPlanes
Maximum number of user-defined clipping planes supported. This member can range from 0 through D3DMAXUSERCLIPPLANES. For a given physical device, this capability might vary across Direct3D Device Objects depending on the parameters supplied to **Direct3D8.CreateDevice**.

MaxVertexBlendMatrices
Maximum number of matrices that this device can apply when performing multimatrix vertex blending. For a given physical device, this capability might vary across Direct3D Device Objects depending on the parameters supplied to **Direct3D8.CreateDevice**.

MaxVertexBlendMatrixIndex
Value that specifies the maximum matrix index that can be indexed into using the per vertex indices. The number of matrices is **MaxVertexBlendMatrixIndex** + 1, which is the size of the matrix palette. If normals are present in the vertex data that needs to be blended for lighting, then the number of matrices is half the number specified by this capability flag. If **MaxVertexBlendMatrixIndex** is set to zero, the driver does not support indexed vertex blending. If this value is not zero then the valid range of indices is zero through **MaxVertexBlendIndexedMatrices**.

A zero value for **MaxVertexBlendMatrixIndex** indicates that the driver does not support indexed matrices.

When software vertex processing is used, 256 matrices could be used for indexed vertex blending, with or without normal blending.

For a given physical device, this capability might vary across Direct3D Device Objects depending on the parameters supplied to **Direct3D8.CreateDevice**.

MaxVertexIndex
Maximum size of indices supported for hardware vertex processing. It is possible to create 32-bit index buffers by specifying D3DFMT_INDEX32; however, you will not be able to render with the index buffer unless this value is greater than &HFFFF.

MaxVertexW
Maximum W-based depth value that the device supports.

MaxVolumeExtent
The maximum volume extent.

MaxVertexShaderConst
Number of vertex shader constant registers.

PixelShaderVersion

The pixel shader version, indicating the level of pixel shader supported by the device. Only pixel shaders with version numbers equal to or less than this will succeed in calls to **Direct3DDevice8.CreatePixelShader**.

- DirectX 8.0 functionality is 01

The main version number is encoded in the second byte. The low byte contains a sub-version number.

PresentationIntervals

A bit mask of values representing what presentation swap intervals are available. Possible values are defined by the **CONST_D3DPRESENT_INTERVAL_FLAGS** enumeration.

PrimitiveMiscCaps

General capabilities for this primitive. Possible values are defined by the **CONST_D3DPMISCCAPSFLAGS** enumeration.

RasterCaps

Information on raster-drawing capabilities. This member can be one or more of the flags defined by the **CONST_D3DPRASTERCAPSFLAGS** enumeration.

ShadeCaps

Shading operations capabilities. It is assumed, in general, that if a device supports a given command at all, it supports the D3DSHADE_FLAT mode (as specified in the **CONST_D3DSHADEMODE** enumeration). This flag specifies whether the driver can also support Gouraud shading and whether alpha color components are supported. When alpha components are not supported, the alpha value of colors generated is implicitly 255. This is the maximum possible alpha (that is, the alpha component is at full intensity).

The color, specular highlights, fog, and alpha interpolants of a triangle each have capability flags that an application can use to find out how they are implemented by the device driver.

This member can be one or more of the flags defined by the **CONST_D3DPSHADECAPSFLAGS** enumeration.

SrcBlendCaps

Source-blending capabilities. This member can be one or more of the flags defined by the **CONST_D3DPBLENDCAPSFLAGS** enumeration.

StencilCaps

Flags specifying supported stencil-buffer operations. Stencil operations are assumed to be valid for all three stencil-buffer operation render states (D3DRS_STENCILFAIL, D3DRS_STENCILPASS, and D3DRS_STENCILFAILZFAIL). Possible values are defined by the **CONST_D3DSTENCILCAPFLAGS** enumeration.

For more information, see the **CONST_D3DSTENCILOP** enumeration.

TextureAddressCaps

Texture-addressing capabilities for a Direct3DTexture object. This member can be one or more of the flags defined by the **CONST_D3DPTADDRESSCAPSFLAGS** enumeration.

TextureCaps

Miscellaneous texture-mapping capabilities. This member can be one or more of the flags defined by the **CONST_D3DPTEXTURECAPSFLAGS** enumeration.

TextureFilterCaps

Texture-filtering capabilities for a Direct3DTexture object. General texture-filtering flags reflect which texture-filtering modes are supported and can be set for the D3DTSS_MAGFILTER, D3DTSS_MINFILTER, or D3DTSS_MIPFILTER texture-stage states. Per-stage filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple-texture blending with the **Direct3DDevice8** class. This member can be any combination of the general and per-stage texture-filtering flags defined by the **CONST_D3DPTFILTERCAPSFLAGS** enumeration.

TextureOpCaps

Combination of flags describing the texture operations supported by this device. Possible values are defined by the **CONST_D3DTEXOPCAPSFLAGS** enumeration.

For more information, see the **CONST_D3DTEXTUREOP** enumeration.

VertexProcessingCaps

Vertex processing capabilities. Possible values are defined by the **CONST_D3DVTXPCAPSFLAGS** enumeration. For a given physical device, this capability might vary across Direct3D Device Objects depending on the parameters supplied to **Direct3D8.CreateDevice**.

VertexShaderVersion

The vertex shader version, indicating the level of vertex shader supported by the device. Only vertex shaders with version numbers equal to or less than this will succeed in calls to **Direct3DDevice8.CreateVertexShader**. The level of shader is specified to **CreateVertexShader** as the first token in the vertex shader token stream.

- DirectX 7.0 functionality is 0
- DirectX 8.0 functionality is 01

The main version number is encoded in the second byte. The low byte contains a sub-version number.

VolumeTextureAddressCaps

Texture-addressing capabilities for Direct3DVolumeTexture objects. This member can be one or more of the flags defined by the **CONST_D3DPTADDRESSCAPSFLAGS** enumeration.

VolumeTextureFilterCaps

Texture-filtering capabilities for a Direct3DVolumeTexture object. Per-stage filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple-texture blending with the **Direct3DDevice8** class. This member can be any combination of the per-stage texture-filtering flags defined for the **TextureFilterCaps** member. This member can be any combination of the general and per-stage texture-filtering flags defined by the **CONST_D3DPTFILTERCAPSFLAGS** enumeration.

ZCmpCaps

Z-buffer comparison capabilities. This member can include the same capability flags defined for the **AlphaCmpCaps** member. The flags for this member are defined by the **CONST_D3DPCMPCAPSFLAGS** enumeration.

Remarks

The **MaxTextureBlendStages** and **MaxSimultaneousTextures** members might seem very similar, but they contain different information. The **MaxTextureBlendStages** member contains the total number of texture-blending stages supported by the current device, and the **MaxSimultaneousTextures** member describes how many of those stages can have textures bound to them by using the **Direct3DDevice8.SetTexture** method.

When the driver fills this type, it can set values for execute-buffer capabilities, even when the class being used to retrieve the capabilities (such as **Direct3DDevice8**) does not support execute buffers.

The following flags concerning mipmapped textures are not supported in DirectX 8.0.

- D3DPTFILTERCAPS_NEAREST
- D3DPTFILTERCAPS_LINEAR
- D3DPTFILTERCAPS_MIPNEAREST
- D3DPTFILTERCAPS_MIPLINEAR
- D3DPTFILTERCAPS_LINEARMIPNEAREST
- D3DPTFILTERCAPS_LINEARMIPLINEAR

See Also

Direct3D8.GetDeviceCaps, **Direct3DDevice8.GetDeviceCaps**

D3DCLIPSTATUS8

#Describes the current clip status.

```
Type D3DCLIPSTATUS8
    ClipUnion As Long
    ClipIntersection As Long
End Type
```

Members

ClipUnion

Clip union flags that describe the current clip status. This member can be one or more of the flags defined by the **CONST_D3DCLIPFLAGS** enumeration.

ClipIntersection

IDH_D3DCLIPSTATUS8_graphicsvb

Clip intersection flags that describe the current clip status. This member can take the same flags as **ClipUnion**.

See Also

Direct3DDevice8.GetClipStatus, **Direct3DDevice8.SetClipStatus**

D3DCOLORVALUE

#Describes color values.

```
Type D3DCOLORVALUE
    a As Single
    b As Single
    g As Single
    r As Single
End Type
```

Members

a, **b**, **g**, and **r**

Values specifying the red, green, blue, and alpha components of a color. These values generally are in the range from 0 through 1, with 0 being black.

Remarks

You can set the members of this type to values outside the range of 0 through 1 to implement some unusual effects. Values greater than 1 produce strong lights that tend to wash out a scene. Negative values produce dark lights, which actually remove light from a scene.

See Also

Color Values for Lights and Materials

D3DDEVICE_CREATION_PARAMETERS

#Describes the creation parameters for a device.

```
Type D3DDEVICE_CREATION_PARAMETERS
    AdapterOrdinal As Long
```

IDH_D3DCOLORVALUE_graphicsvb
IDH_D3DDEVICE_CREATION_PARAMETERS_graphicsvb

```
    BehaviorFlags As Long
    DeviceType As CONST_D3DDEVTYPE
    hFocusWindow As Long
End Type
```

Members

AdapterOrdinal

Ordinal number that denotes the display adapter. D3DADAPTER_DEFAULT is always the primary display adapter.

Use this ordinal as the *Adapter* parameter for any of the **Direct3D8** methods. Note that different Direct3D8 objects may use different ordinals. For example, adapters can enter and leave a system due to users adding or subtracting monitors from a multiple monitor system, or due to hot-swapping a laptop. Consequently, you should use this ordinal only in a **Direct3D8** known to be valid. The only two valid **Direct3D8** instances are the **Direct3D8** that created this **Direct3DDevice8** object and the **Direct3D8** returned from **Direct3DDevice8.GetDirect3D** as called through this **Direct3DDevice8** object.

BehaviorFlags

A combination of one or more of the flags that control global behaviors of the Microsoft® Direct3D® device. Possible values are defined by the **CONST_D3DCREATEFLAGS** enumeration.

The value of this parameter mirrors the value passed to the **Direct3D8.CreateDevice** call that created this device.

DeviceType

Member of the **CONST_D3DDEVTYPE** enumeration. Denotes the amount of emulated functionality for this device. The value of this parameter mirrors the value passed to the **CreateDevice** call that created this device.

hFocusWindow

Window handle to which focus belongs for this Direct3D device. The value of this parameter mirrors the value passed to the **CreateDevice** call that created this device.

See Also

Direct3DDevice8.GetCreationParameters, **Direct3D8.CreateDevice**

D3DDISPLAYMODE

#Describes the display mode.

```
Type D3DDISPLAYMODE
    Format As CONST_D3DFORMAT
    Height As Long
```

IDH_D3DDISPLAYMODE_graphicsvb

```
    RefreshRate As Long
    Width As Long
End Type
```

Members

Format
A member of the **CONST_D3DFORMAT** enumeration, describing the surface format of the display mode.

Height
The screen height, in pixels.

RefreshRate
The refresh rate. The value of 0 indicates an adapter default.

Width
The screen width, in pixels.

See Also

Direct3D8.EnumAdapterModes, **Direct3D8.GetAdapterDisplayMode**, **Direct3DDevice8.GetDisplayMode**

D3DGAMMARAMP

#Contains red, green, and blue ramp data.

```
Type D3DGAMMARAMP
    blue(0 To 255) As Integer
    green(0 To 255) As Integer
    red(0 To 255) As Integer
End Type
```

Members

blue, **green**, and **red**
Array of 256 **Integer** elements that describe the red, green, and blue gamma ramps.

See Also

Direct3DDevice8.GetGammaRamp, **Direct3DDevice8.SetGammaRamp**

IDH_D3DGAMMARAMP_graphicsvb

D3DINDEXBUFFER_DESC

#Describes an index buffer.

```
Type D3DINDEXBUFFER_DESC
   Format As CONST_D3DFORMAT
   Pool As CONST_D3DPOOL
   Size As Long
   Type As CONST_D3DRESOURCETYPE
   Usage As Long
End Type
```

Members

Format
A member of the **CONST_D3DFORMAT** enumeration, describing the surface format of the index buffer data.

Pool
A member of the **CONST_D3DPOOL** enumeration, specifying the class of memory allocated for this index buffer.

Size
The size of the index buffer, in bytes.

Type
A member of the **CONST_D3DRESOURCETYPE** enumeration, identifying this resource as an index buffer.

Usage
A combination of one or more of the following flags defined by the **CONST_D3DUSAGEFLAGS** enumeration, specifying the usage for this resource.

D3DUSAGE_DONOTCLIP
Set to indicate that the index buffer content will never require clipping.

D3DUSAGE_HOSURFACES
Set to indicate when the index buffer is to be used for drawing high-order primitives.

D3DUSAGE_RTPATCHES
Set to indicate when the vertex buffer is to be used for drawing high-order primitives.

D3DUSAGE_NPATCHES
Set to indicate when the index buffer is to be used for drawing N patches.

D3DUSAGE_POINTS
Set to indicate when the index buffer is to be used for drawing point sprites or indexed point lists.

D3DUSAGE_SOFTWAREPROCESSING

IDH_D3DINDEXBUFFER_DESC_graphicsvb

Set to indicate that the buffer is to be used with software processing.

D3DUSAGE_WRITEONLY
Informs the system that the application writes only to the index buffer. Using this flag enables the driver to choose the best memory location for efficient write operations and rendering. Attempts to read from an index buffer that is created with this capability can result in degraded performance.

See Also

Direct3DIndexBuffer8.GetDesc

D3DLIGHT8

#Defines a set of lighting properties.

```
Type D3DLIGHT8
    ambient As D3DCOLORVALUE
    diffuse As D3DCOLORVALUE
    specular As D3DCOLORVALUE
    type As CONST_D3DLIGHTTYPE
    Attenuation0 As Single
    Attenuation1 As Single
    Attenuation2 As Single
    Direction As D3DVECTOR
    Falloff As Single
    Phi As Single
    position As D3DVECTOR
    Range As Single
    Theta As Single
End Type
```

Members

ambient
Ambient color emitted by the light. This member is a **D3DCOLORVALUE** type.

diffuse
Diffuse color emitted by the light. This member is a **D3DCOLORVALUE** type.

specular
Specular color emitted by the light. This member is a **D3DCOLORVALUE** type.

type

IDH_D3DLIGHT8_graphicsvb

Type of the light source. This value is one of the members of the **CONST_D3DLIGHTTYPE** enumeration.

Attenuation0, **Attenuation1**, and **Attenuation2**
Values specifying how the light intensity changes over distance. Attenuation values are ignored for directional lights. These members represent attenuation constants. For information on attenuation, see Light Attenuation Over Distance. Valid values for these members range from 0.0 to infinity. For non-directional lights, all three attenuation values should not be set to 0.0 at the same time.

Direction
Direction that the light is pointing in world space, specified by a **D3DVECTOR** type. This member has meaning only for directional and spotlights. This vector need not be normalized, but it should have a nonzero length.

Falloff
Decrease in illumination between a spotlight's inner cone (the angle specified by **Theta**) and the outer edge of the outer cone (the angle specified by **Phi**).

The effect of falloff on the lighting is subtle. Furthermore, a small performance penalty is incurred by shaping the falloff curve. For these reasons, most developers set this value to 1.0.

Phi
Angle, in radians, defining the outer edge of the spotlight's outer cone. Points outside this cone are not lit by the spotlight. This value must be between 0 and *pi*.

position
Position of the light in world space, specified by a **D3DVECTOR** type. This member has no meaning for directional lights and is ignored in that case.

Range
Distance beyond which the light has no effect. The maximum allowable value for this member is the square root of FLT_MAX. This member does not affect directional lights.

Theta
Angle, in radians, of a spotlight's inner cone—that is, the fully illuminated spotlight cone. This value must be in the range from 0 through the value specified by **Phi**.

See Also

Direct3DDevice8.GetLight, **Direct3DDevice8.SetLight**

D3DLINEPATTERN

#Describes a line pattern.

```
Type D3DLINEPATTERN
   LinePattern As Integer
```

IDH_D3DLINEPATTERN_graphicsvb

```
    RepeatFactor As Integer
End Type
```

Members

LinePattern
Bits specifying the line pattern. For example, the following value would produce a dotted line: 1100110011001100.

RepeatFactor
Number of times to repeat each series of 1s and 0s specified in the **LinePattern** member. This enables an application to stretch the line pattern.

Remarks

These values are used by the D3DRS_LINEPATTERN render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

A line pattern specifies how a line is drawn. The line pattern is always the same, no matter where it is started. (This differs from stippling, which affects how objects are rendered; that is, to imitate transparency.)

The line pattern specifies up to a 16-pixel pattern of on and off pixels along the line. The **RepeatFactor** member specifies how many pixels are repeated for each entry in **LinePattern**.

D3DLOCKED_BOX

#Describes a locked box (volume).

```
Type D3DLOCKED_BOX
    pBits As Long
    RowPitch As Long
    SlicePitch As Long
End Type
```

Members

pBits
The locked bits. If a D3DBOX was provided to the **LockRect** call, *pBits* will be appropriately offset from the start of the volume.

RowPitch
The row pitch, in bytes.

SlicePitch
The slice pitch, in bytes.

IDH_D3DLOCKED_BOX_graphicsvb

Remarks

Volumes can be visualized as being organized into slices of *width* x *height* with 2-D surfaces stacked up to make a *width* x *height* x *depth* volume. For more information, see Volume Texture Resources.

See Also

Direct3DVolume8.LockBox, **Direct3DVolumeTexture8.LockBox**

D3DLOCKED_RECT

#Describes a locked rectangular region.

```
Type D3DLOCKED_RECT
    pBits As Long
    Pitch As Long
End Type
```

Members

pBits
The locked bits. If a RECT was provided to the **LockRect** call, *pBits* will be appropriately offset from the start of the surface.

Pitch
Pitch of surface, in bytes.

See Also

Direct3DCubeTexture8.LockRect, **Direct3DSurface8.LockRect**, **Direct3DTexture8.LockRect**.

D3DLVERTEX

#Defines an untransformed and lit vertex (model coordinates with color).

```
Type D3DLVERTEX
    color As Long
    specular As Long
    tu As Single
    tv As Single
    x As Single
    y As Single
    z As Single
```

IDH_D3DLOCKED_RECT_graphicsvb
IDH_D3DLVERTEX_graphicsvb

```
End Type
```

Members

color and **specular**
Values describing the color and specular component of the vertex.

tu and **tv**
Values describing the texture coordinates of the vertex.

x, **y**, and **z**
Values describing the homogeneous coordinates of the vertex.

Remarks

An application should use this type when the vertex transformations will be handled by Microsoft® Direct3D®. This type contains only data and a color that would be filled by software lighting.

If D3DRS_SPECULARENABLE is TRUE, the specular component will be added to the base color after the texture cascade but before alpha blending. However, you can assign the specular component to be applied during texture color processing by setting the D3DTA_SPECULAR flag. For more information, see Texture Argument Flags.

D3DLVERTEX2

#Defines an untransformed and lit vertex (model coordinates with color) with two sets of texture coordinates.

```
Type D3DLVERTEX2
    color As Long
    specular As Long
    tu1 As Single
    tu2 As Single
    tv1 As Single
    tv2 As Single
    x As Single
    y As Single
    z As Single
End Type
```

Members

color and **specular**
Values describing the color and specular component of the vertex.

IDH_D3DLVERTEX2_graphicsvb

tu1 and **tv1**
Values describing the first set of texture coordinates of the vertex.

tu2 and **tv2**
Values describing the second set of texture coordinates of the vertex.

x, **y**, and **z**
Values describing the homogeneous coordinates of the vertex.

Remarks

D3DLVERTEX2 is provided as a convenience, since many multitexture operations require two sets of texture coordinates.

An application should use this type when the vertex transformations will be handled by Microsoft® Direct3D®. This type contains only data and a color that would be filled by software lighting.

If D3DRS_SPECULARENABLE is TRUE, the specular component will be added to the base color after the texture cascade but before alpha blending. However, you can assign the specular component to be applied during texture color processing by setting the D3DTA_SPECULAR flag. For more information, see Texture Argument Flags.

D3DMATERIAL8

#Specifies material properties.

```
Type D3DMATERIAL8
    ambient As D3DCOLORVALUE
    diffuse As D3DCOLORVALUE
    emissive As D3DCOLORVALUE
    power As Single
    specular As D3DCOLORVALUE
End Type
```

Members

ambient, **diffuse**, **emissive**, and **specular**
Values specifying the ambient color, diffuse color, emissive color, and specular color of the material, respectively. These values are **D3DCOLORVALUE** types.

power
Value specifying the sharpness of specular highlights. To turn off specular highlights for a material, set this member to 0; setting the specular color components to 0 is not enough.

IDH_D3DMATERIAL8_graphicsvb

See Also

Direct3DDevice8.GetMaterial, **Direct3DDevice8.SetMaterial**

D3DMATRIX

#Describes a matrix.

```
Type D3DMATRIX
    m11 As Single
    m12 As Single
    m13 As Single
    m14 As Single
    m21 As Single
    m22 As Single
    m23 As Single
    m24 As Single
    m31 As Single
    m32 As Single
    m33 As Single
    m34 As Single
    m41 As Single
    m42 As Single
    m43 As Single
    m44 As Single
End Type
```

Remarks

In Microsoft® Direct3D®, the **m34** element of a projection matrix cannot be a negative number. If an application needs to use a negative value in this location, it should scale the entire projection matrix by –1, instead.

See Also

Direct3DDevice8.GetTransform, **Direct3DDevice8.MultiplyTransform**, **Direct3DDevice8.SetTransform**

D3DPLANE

#Describes a plane.

```
Type D3DPLANE
    a As Single
```

IDH_D3DMATRIX_graphicsvb
IDH_D3DPLANE_graphicsvb

```
    b As Single
    c As Single
    d As Single
End Type
```

Members

a, **b**, **c**, and **d**
The a, b, c, and d coefficients, respectively, of the clipping plane in the general plane equation. See Remarks.

Remarks

The members of the **D3DPLANE** type take the form of the general plane equation. They fit into the general plane equation so that ax + by + cz + dw = 0.

D3DPRESENT_PARAMETERS

#Describes the presentation parameters.

```
Type D3DPRESENT_PARAMETERS
    AutoDepthStencilFormat As CONST_D3DFORMAT
    BackBufferCount As Long
    BackBufferFormat As CONST_D3DFORMAT
    BackBufferHeight As Long
    BackBufferWidth As Long
    EnableAutoDepthStencil As Long
    Flags As Long
    FullScreen_PresentationInterval As Long
    FullScreen_RefreshRateInHz As Long
    hDeviceWindow As Long
    MultiSampleType As CONST_D3DMULTISAMPLE_TYPE
    SwapEffect As CONST_D3DSWAPEFFECT
    Windowed As Long
End Type
```

Members

AutoDepthStencilFormat
A member of the **CONST_D3DFORMAT** enumeration. The format of the automatic depth-stencil surface that the device will create. This member is ignored unless **EnableAutoDepthStencil** is a non-zero value (TRUE).

BackBufferCount

IDH_D3DPRESENT_PARAMETERS_graphicsvb

This value can be 0, 1, 2 or 3. Note that 0 is treated as 1. If the number of back buffers cannot be created, Microsoft® Direct3D will fail the method call and fill this value with the number of back buffers that could be created. As a result, an application can call the method twice with the same **D3DPRESENT_PARAMETERS** type and expect it to work the second time.

One back buffer is considered the minimum number of back buffers. The method call fails if 1 back buffer cannot be created. The value of **BackBufferCount** influences what set of swap effects are allowed. Specifically, any D3DSWAPEFFECT_COPY swap effect requires that there be exactly one back buffer.

BackBufferFormat

A member of the **CONST_D3DFORMAT** enumeration. This value must be one of the render target formats as validated by **Direct3D8.CheckDeviceType**.

If **Windowed** is set to TRUE, then **BackBufferFormat** must be set to match the format of the current display mode. Use **Direct3DDevice8.GetDisplayMode** to obtain the current format.

BackBufferHeight and **BackBufferWidth**

The height and width of the new swap chain's back buffers, in pixels. If **Windowed** is zero (FALSE) the presentation is full-screen, and these values must equal the height and width of one of the enumerated display modes found through **Direct3D8.EnumAdapterModes**. If **Windowed** is TRUE and either of these values is zero, then the corresponding dimension of the client area of the **hDeviceWindow** (or the focus window, if **hDeviceWindow** is Nothing) is taken.

EnableAutoDepthStencil

If this value is TRUE, Direct3D® will manage depth buffers for the application. The device will create a depth-stencil buffer when it is created. The depth-stencil buffer will be automatically set as the render target of the device. When the device is reset, the depth-stencil buffer will be automatically destroyed and recreated in the new size.

If **EnableAutoDepthStencil** is TRUE, then **AutoDepthStencilFormat** must be a valid depth-stencil format.

Flags

Currently this value is not used, it must be set to 0.

FullScreen_PresentationInterval

A bit mask of values representing what presentation swap intervals are available. For windowed mode, this value must be 0; otherwise, this value must be one of the values enumerated in the **PresentationIntervals** member of **D3DCAPS8**. Possible values are defined by the **CONST_D3DPRESENT_INTERVAL_FLAGS** enumeration.

FullScreen_RefreshRateInHz

The rate at which the display adapter refreshes the screen. For windowed mode, this value must be 0; otherwise, this value must be one of the refresh rates returned by **Direct3D8.EnumAdapterModes**, or one of the refresh rates defined by the **CONST_D3DPRESENT_RATE_FLAGS** enumeration.

hDeviceWindow

If full-screen, this is the cover window. If windowed, this will be the default target window for **Direct3DDevice8.Present**. If this value is Nothing, the focus window will be taken. For applications that use multiple full-screen devices, such as a multi-monitor system, exactly one device should use the focus window as the device window. All other devices should have unique device windows. Otherwise, behavior is undefined and applications will not work as expected.

Note that no attempt is made by Direct3D to reflect user changes in window size. The back buffer is not implicitly reset when this window is reset. However, the **Present** method does automatically track window position changes.

MultiSampleType

A member of the **CONST_D3DMULTISAMPLE_TYPE** enumeration. The value must be D3DMULTISAMPLE_NONE unless **SwapEffect** has been set to D3DSWAPEFFECT_DISCARD. Multisampling is supported only if the swap effect is D3DSWAPEFFECT_DISCARD.

SwapEffect

A member of the **CONST_D3DSWAPEFFECT** enumeration. Direct3D will guarantee the implied semantics concerning buffer swap behavior. So if **Windowed** is TRUE and D3DSWAPEFFECT_FLIP, then Direct3D will create one extra back buffer, and copy whichever becomes the front buffer at presentation time.

D3DSWAPEFFECT_COPY and D3DSWAPEFFECT_COPY_VSYNC require that **BackBufferCount** be set to 1.

D3DSWAPEFFECT_DISCARD will be enforced in the debug run time by filling any buffer with noise after it is presented.

Windowed

TRUE if the application runs windowed, FALSE if the application runs full-screen.

See Also

Direct3D8.CreateDevice, **Direct3DDevice8.CreateAdditionalSwapChain**, **Direct3DDevice8.Present**, **Direct3DDevice8.Reset**

D3DQUATERNION

#Describes a quaternion.

```
Type D3DQUATERNION
   w As Single
   x As Single
   y As Single
   z As Single
End Type
```

IDH_D3DQUATERNION_graphicsvb

Members

w, **x**, **y**, and **z**
The w-, x-, y-, and z-components, respectively, of the quaternion.

Remarks

Quaternions add a fourth element to the [*x, y, z*] values that define a vector, resulting in arbitrary 4-D vectors. However, the following illustrates how each element of a unit quaternion relates to an axis-angle rotation (where *q* represents a unit quaternion (x, y, z, w), *axis* is normalized, and *theta* is the desired CCW rotation about the axis):

```
q.x = Sin(theta / 2) * axis.x
q.y = Sin(theta / 2) * axis.y
q.z = Sin(theta / 2) * axis.z
q.w = Cos(theta / 2)
```

D3DRANGE

#Defines a range.

```
Type D3DRANGE
   cbOffset As Long
   size As Long
End Type
```

Members

cbOffset
The offset, in bytes.

size
The size, in bytes.

D3DRASTER_STATUS

#Describes the raster status.

```
Type D3DRASTER_STATUS
   InVBLANK As Long
   ScanLine As Long
End Type
```

IDH_D3DRANGE_graphicsvb
IDH_D3DRASTER_STATUS_graphicsvb

Members

InVBLANK
TRUE if the raster is in the vertical blank period. FALSE if the raster is not in the vertical blank period.

ScanLine
If **InVBLANK** is FALSE, then this value is an integer roughly corresponding to the current scan line painted by the raster. Scan lines are numbered in the same way as Microsoft® Direct3D® surface coordinates: 0 is the top of the primary surface, extending to the value (height of the surface - 1) at the bottom of the display.

If **InVBLANK** is TRUE, then this value is set to zero and can be ignored.

D3DRECT

#Defines a rectangle.

```
Type D3DRECT
   x1 As Long
   x2 As Long
   y1 As Long
   y2 As Long
End Type
```

Members

x1 and **y1**
Coordinates of the upper-left corner of the rectangle.

x2 and **y2**
Coordinates of the lower-right corner of the rectangle.

See Also

Direct3DDevice8.Clear

D3DRECTPATCH_INFO

#Describes a rectangular high-order patch.

```
Type D3DRECTPATCH_INFO
   Basis As CONST_D3DBASISTYPE
```

IDH_D3DRECT_graphicsvb
IDH_D3DRECTPATCH_INFO_graphicsvb

```
    Height As Long
    Order As CONST_D3DORDERTYPE
    StartVertexOffsetHeight As Long
    StartVertexOffsetWidth As Long
    StrideBytes As Long
    Width As Long
End Type
```

Members

Basis
Member of the **CONST_D3DBASISTYPE** type, defining the basis type for the rectangular high-order patch.

Height
Height of each vertex, in number of vertices.

Order
Member of the **CONST_D3DORDERTYPE** type, defining the order type for the rectangular high-order patch.

StartVertexOffsetHeight
Starting vertex offset height, in number of vertices.

StartVertexOffsetWidth
Starting vertex offset width, in number of vertices.

StrideBytes
Stride between segments, in number of vertices.

Width
Width of each vertex, in number of vertices.

Remarks

To render a stream of individual rectangular patches (non-mosaic), you should interpret your geometry as a long narrow (1 × N) rectangular patch. The **D3DRECTPATCH_INFO** structure for such a strip (cubic bézier) would be set up in the following manner.

```
Dim i As Integer
Dim RectInfo As D3DRECTPATCH_INFO

With D3DRECTPATCH_INFO
   .Width = 4
   .Height = 4
   .Stride = 4
   .Basis = D3DBASIS_BEZIER
   .Order = D3DORDER_CUBIC
   .StartVertexOffsetWidth = 0
   .StartVertexOffsetHeight = 4 * i  ' The variable i is the index of the patch you want to render.
```

```
End With
```

See Also

Direct3DDevice8.DrawRectPatch

D3DSURFACE_DESC

#Describes a surface.

```
Type D3DSURFACE_DESC
    format As CONST_D3DFORMAT
    Height As Long
    MultiSampleType As CONST_D3DMULTISAMPLE_TYPE
    Pool As CONST_D3DPOOL
    size As Long
    type As CONST_D3DRESOURCETYPE
    Usage As Long
    Width As Long
End Type
```

Members

format
A member of the **CONST_D3DFORMAT** enumeration, describing the surface format.

Height
The height of the surface, in pixels.

MultiSampleType
A member of the **CONST_D3DMULTISAMPLE_TYPE** enumeration, specifying the levels of full-scene multisampling supported by the surface.

Pool
A member of the **CONST_D3DPOOL** enumeration, specifying the class of memory allocated for this surface.

size
The size of the surface, in bytes.

type
A member of the **CONST_D3DRESOURCETYPE** enumerated type, identifying this resource as a surface.

Usage
A combination of one or more of the following flags defined by the **CONST_D3DUSAGEFLAGS** enumeration, specifying the usage for this resource.

IDH_D3DSURFACE_DESC_graphicsvb

D3DUSAGE_DEPTHSTENCIL
Set to indicate that the surface is to be used as a depth stencil surface.

D3DUSAGE_RENDERTARGET
Set to indicate that the surface is to be used as a render target.

Width
The width of the surface, in pixels.

See Also

Direct3DCubeTexture8.GetLevelDesc, **Direct3DSurface8.GetDesc**, **Direct3DTexture8.GetLevelDesc**.

D3DTLVERTEX

#Defines a transformed and lit vertex (screen coordinates with color).

```
Type D3DTLVERTEX
    color As Long
    rhw As Single
    specular As Long
    sx As Single
    sy As Single
    sz As Single
    tu As Single
    tv As Single
End Type
```

Members

color and **specular**
Values describing the color and specular component of the vertex.

rhw
Value that is the reciprocal of homogeneous w from homogeneous coordinates (x,y,z,w). This value is often 1 divided by the distance from the origin to the object along the z-axis.

sx, **sy**, and **sz**
Values describing a vertex in screen coordinates. The largest allowable value for **dvSZ** is 1.0 if you want the vertex to be within the range of z-values that are displayed.

tu and **tv**
Values describing the texture coordinates of the vertex.

IDH_D3DTLVERTEX_graphicsvb

Remarks

Microsoft® Direct3D® uses the current viewport parameters (the **x**, **y**, **Width**, and **Height** members of the **D3DVIEWPORT8** type) to clip **D3DTLVERTEX** vertices. The system always clips z-coordinates to [0, 1].

If D3DRS_SPECULARENABLE is TRUE, the specular component will be added to the base color after the texture cascade but before alpha blending. However, you can assign the specular component to be applied during texture color processing by setting the D3DTA_SPECULAR flag. For more information, see Texture Argument Flags.

See Also

D3DTLVERTEX2, **D3DVERTEX**, **D3DVERTEX2**

D3DTLVERTEX2

#Defines a transformed and lit vertex (screen coordinates with color) with two sets of texture coordinates.

```
Type D3DTLVERTEX2
    color As Long
    rhw As Single
    specular As Long
    sx As Single
    sy As Single
    sz As Single
    tu1 As Single
    tu2 As Single
    tv1 As Single
    tv2 As Single
End Type
```

Members

color and **specular**
Values describing the color and specular component of the vertex.

rhw
Value that is the reciprocal of homogeneous w from homogeneous coordinates (x,y,z,w). This value is often 1 divided by the distance from the origin to the object along the z-axis.

sx, **sy**, and **sz**

IDH_D3DTLVERTEX2_graphicsvb

Values describing a vertex in screen coordinates. The largest allowable value for **dvSZ** is 1.0 if you want the vertex to be within the range of z-values that are displayed.

tu1 and **tv1**
Values describing the first set of texture coordinates of the vertex.

tu2 and **tv2**
Values describing the second set of texture coordinates of the vertex.

Remarks

D3DTLVERTEX2 is provided as a convenience, since many multitexture operations require two sets of texture coordinates.

Microsoft® Direct3D® uses the current viewport parameters (the **x**, **y**, **Width**, and **Height** members of the **D3DVIEWPORT8** type) to clip **D3DTLVERTEX** vertices. The system always clips z-coordinates to [0, 1].

If D3DRS_SPECULARENABLE is TRUE, the specular component will be added to the base color after the texture cascade but before alpha blending. However, you can assign the specular component to be applied during texture color processing by setting the D3DTA_SPECULAR flag. For more information, see Texture Argument Flags.

See Also

D3DVERTEX, **D3DVERTEX2**, **D3DTLVERTEX**

D3DTRIPATCH_INFO

#Describes a triangular high-order patch.

```
Type D3DTRIPATCH_INFO
   Basis As CONST_D3DBASISTYPE
   Order As CONST_D3DORDERTYPE
   StartVertexOffsetHeight As Long
   StartVertexOffsetWidth As Long
End Type
```

Members

Basis
Member of the **CONST_D3DBASISTYPE** type, defining the basis type for the triangular high-order patch.

Order

IDH_D3DTRIPATCH_INFO_graphicsvb

Member of the **CONST_D3DORDERTYPE** type, defining the order type for the triangular high-order patch.

StartVertexOffsetHeight
Starting vertex offset height, in number of vertices.

StartVertexOffsetWidth
Starting vertex offset width, in number of vertices.

See Also

Direct3DDevice8.DrawTriPatch

D3DVECTOR

#Defines a vector in three-dimensional space.

```
Type D3DVECTOR
   x As Single
   y As Single
   z As Single
End Type
```

Members

x, **y**, and **z**
Values describing the vector.

See Also

D3DVECTOR2, **D3DVECTOR4**

D3DVECTOR2

#Describes a vector in two-dimensional (2-D) space.

```
Type D3DVECTOR2
   x As Single
   y As Single
End Type
```

Members

x and **y**

IDH_D3DVECTOR_graphicsvb
IDH_D3DVECTOR2_graphicsvb

The x- and y-component, respectively, of the vector.

See Also

D3DVECTOR, **D3DVECTOR4**

D3DVECTOR4

#Describes a vector in four-dimensional (4-D) space.

```
Type D3DVECTOR4
   w As Single
   x As Single
   y As Single
   z As Single
End Type
```

Members

w, **x**, **y**, and **z**
The w-, x-, y-, and z- component, respectively, of the vector.

See Also

D3DVECTOR, **D3DVECTOR2**

D3DVERTEX

#Defines an untransformed and unlit vertex (model coordinates with normal direction vector).

```
Type D3DVERTEX
   nx As Single
   ny As Single
   nz As Single
   tu As Single
   tv As Single
   x As Single
   y As Single
   z As Single
End Type
```

IDH_D3DVECTOR4_graphicsvb
IDH_D3DVERTEX_graphicsvb

Members

nx, **ny**, and **nz**
Values describing the normal coordinates of the vertex.

tu and **tv**
Values describing the texture coordinates of the vertex.

x, **y**, and **z**
Values describing the homogeneous coordinates of the vertex.

See Also

D3DTLVERTEX**D3DVERTEX2**

#Defines an untransformed and unlit vertex (model coordinates with normal direction vector) with two sets of texture coordinates.

```
Type D3DVERTEX2
    nx As Single
    ny As Single
    nz As Single
    tu1 As Single
    tu2 As Single
    tv1 As Single
    tv2 As Single
    x As Single
    y As Single
    z As Single
End Type
```

Members

nx, **ny**, and **nz**
Values describing the normal coordinates of the vertex.

tu1 and **tv1**
Values describing the first set of texture coordinates of the vertex.

tu2 and **tv2**
Values describing the second set of texture coordinates of the vertex.

x, **y**, and **z**
Values describing the homogeneous coordinates of the vertex.

IDH_D3DVERTEX2_graphicsvb

Remarks

D3DVERTEX2 is provided as a convenience, since many multitexture operations require two sets of texture coordinates.

See Also

D3DVERTEX, D3DTLVERTEX, D3DTLVERTEX2**D3DVERTEXBUFFER_DESC**

#Describes a vertex buffer.

```
Type D3DVERTEXBUFFER_DESC
   format As CONST_D3DFORMAT
   FVF As Long
   Pool As CONST_D3DPOOL
   size As Long
   type As CONST_D3DRESOURCETYPE
   Usage As Long
End Type
```

Members

format
A member of the **CONST_D3DFORMAT** enumeration, describing the surface format of the vertex buffer data.

FVF
A combination of flexible vertex format flags that describes the vertex format of the vertices in this buffer.

Pool
A member of the **CONST_D3DPOOL** enumeration, specifying the class of memory allocated for this vertex buffer.

size
The size of the vertex buffer, in bytes

type
A member of the **CONST_D3DRESOURCETYPE** enumeration, identifying this resource as a vertex buffer.

Usage
A combination of one or more of the following flags defined by the **CONST_D3DUSAGEFLAGS** enumeration, specifying the usage for this resource.

IDH_D3DVERTEXBUFFER_DESC_graphicsvb

D3DUSAGE_DONOTCLIP
Set to indicate that the vertex buffer content will never require clipping.

D3DUSAGE_HOSURFACES
Set to indicate when the vertex buffer is to be used for drawing high-order primitives.

D3DUSAGE_RTPATCHES
Set to indicate when the vertex buffer is to be used for drawing high-order primitives.

D3DUSAGE_NPATCHES
Set to indicate when the vertex buffer is to be used for drawing N patches.

D3DUSAGE_POINTS
Set to indicate when the vertex buffer is to be used for drawing point sprites or indexed point lists.

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the vertex buffer is to be used with software vertex processing.

D3DUSAGE_WRITEONLY
Informs the system that the application writes only to the vertex buffer. Using this flag enables the driver to choose the best memory location for efficient write operations and rendering. Attempts to read from a vertex buffer that is created with this capability can result in degraded performance.

See Also

Direct3DVertexBuffer8.GetDesc

D3DVIEWPORT8

#Defines the window dimensions of a render target surface onto which a 3-D volume projects.

```
Type D3DVIEWPORT8
    MaxZ As Single
    MinZ As Single
    Height As Long
    Width As Long
    x As Long
    y As Long
End Type
```

Members

MaxZ and **MinZ**

IDH_D3DVIEWPORT8_graphicsvb

Values describing the range of depth values into which a scene is to be rendered, the minimum and maximum values of the clip volume. Most applications set these values to 0.0 and 1.0, respectively. Clipping is performed after applying the projection matrix.

Height and **Width**
Dimensions of the viewport on the render target surface, in pixels. Unless you are rendering only to a subset of the surface, these members should be set to the dimensions of the render target surface.

x and **y**
Pixel coordinates of the upper-left corner of the viewport on the render target surface. Unless you want to render to a subset of the surface, these members can be set to 0.

Remarks

The **x**, **y**, **Width**, and **Height** members describe the position and dimensions of the viewport on the render target surface. Usually, applications render to the entire target surface; when rendering on a 640×480 surface, these members should be 0, 0, 640, and 480, respectively. The **MinZ** and **MaxZ** are typically set to 0.0 and 1.0 but can be set to other values to achieve specific effects. For example, you might set them both to 0.0 to force the system to render objects to the foreground of a scene, or both to 1.0 to force the objects into the background.

When the viewport parameters for a device change (due to a call to the **Direct3DDevice8.SetViewport** method), the driver builds a new transformation matrix.

See Also

Direct3DDevice8.GetViewport, **Direct3DDevice8.SetViewport**

D3DVOLUME_DESC

#Describes a volume.

```
Type D3DVOLUME_DESC
    Depth As Long
    format As CONST_D3DFORMAT
    Height As Long
    Pool As CONST_D3DPOOL
    size As Long
    type As CONST_D3DRESOURCETYPE
    Usage As Long
    Width As Long
End Type
```

IDH_D3DVOLUME_DESC_graphicsvb

Members

Depth
The depth of the volume, in pixels.

format
A member of the **CONST_D3DFORMAT** enumeration, describing the surface format of the volume.

Height
The height of the volume, in pixels.

Pool
A member of the **CONST_D3DPOOL** enumeration, specifying the class of memory allocated for this volume.

size
The size of the volume, in bytes.

type
A member of the **CONST_D3DRESOURCETYPE** enumeration, identifying this resource as a volume.

Usage
Currently not used. Will always be returned as 0.

Width
The width of the volume, in pixels.

See Also

Direct3DVolume8.GetDesc, **Direct3DVolumeTexture8.GetLevelDesc**

Enumerations

This section contains information about the following enumerations used with Microsoft® Direct3D®.

- **CONST_D3DBACKBUFFERTYPE**
- **CONST_D3DBASISTYPE**
- **CONST_D3DBLEND**
- **CONST_D3DBLENDOP**
- **CONST_D3DCAPS2FLAGS**
- **CONST_D3DCAPSFLAGS**
- **CONST_D3DCLEARFLAGS**

- **CONST_D3DCLIPFLAGS**
- **CONST_D3DCLIPPLANEFLAGS**
- **CONST_D3DCMPFUNC**
- **CONST_D3DCOLORWRITEENABLEFLAGS**
- **CONST_D3DCONST**
- **CONST_D3DCREATEFLAGS**
- **CONST_D3DCUBEMAP_FACES**
- **CONST_D3DCULL**
- **CONST_D3DCURSORCAPSFLAGS**
- **CONST_D3DDEBUGMONITORTOKENS**
- **CONST_D3DDEVCAPSFLAGS**
- **CONST_D3DDEVTYPE**
- **CONST_D3DERR**
- **CONST_D3DFILLMODE**
- **CONST_D3DFOGMODE**
- **CONST_D3DFORMAT**
- **CONST_D3DFVFCAPSFLAGS**
- **CONST_D3DFVFFLAGS**
- **CONST_D3DFVFTEXTUREFORMATS**
- **CONST_D3DLIGHTTYPE**
- **CONST_D3DLINECAPS**
- **CONST_D3DLOCKFLAGS**
- **CONST_D3DMATERIALCOLORSOURCE**
- **CONST_D3DMULTISAMPLE_TYPE**
- **CONST_D3DORDERTYPE**
- **CONST_D3DPATCHEDGESTYLE**
- **CONST_D3DPBLENDCAPSFLAGS**
- **CONST_D3DPCMPCAPSFLAGS**
- **CONST_D3DPMISCCAPSFLAGS**
- **CONST_D3DPOOL**
- **CONST_D3DPRASTERCAPSFLAGS**
- **CONST_D3DPRESENT_INTERVAL_FLAGS**
- **CONST_D3DPRESENT_RATE_FLAGS**
- **CONST_D3DPRIMITIVETYPE**
- **CONST_D3DPSHADECAPSFLAGS**
- **CONST_D3DPTADDRESSCAPSFLAGS**
- **CONST_D3DPTEXTURECAPSFLAGS**

- **CONST_D3DPTFILTERCAPSFLAGS**
- **CONST_D3DRENDERSTATETYPE**
- **CONST_D3DRESOURCETYPE**
- **CONST_D3DSCPFLAGS**
- **CONST_D3DSGRFLAGS**
- **CONST_D3DSHADEMODE**
- **CONST_D3DSPDFLAGS**
- **CONST_D3DSTATEBLOCKTYPE**
- **CONST_D3DSTENCILCAPFLAGS**
- **CONST_D3DSTENCILOP**
- **CONST_D3DSWAPEFFECT**
- **CONST_D3DTAFLAGS**
- **CONST_D3DTEXOPCAPSFLAGS**
- **CONST_D3DTEXTUREADDRESS**
- **CONST_D3DTEXTUREFILTERTYPE**
- **CONST_D3DTEXTUREOP**
- **CONST_D3DTEXTURESTAGESTATETYPE**
- **CONST_D3DTEXTURETRANSFORMFLAGS**
- **CONST_D3DTRANSFORMSTATETYPE**
- **CONST_D3DTSS_TCI_FLAGS**
- **CONST_D3DUSAGEFLAGS**
- **CONST_D3DVERTEXBLENDFLAGS**
- **CONST_D3DVTXPCAPSFLAGS**
- **CONST_D3DWRAPBIAS**
- **CONST_D3DWRAPFLAGS**
- **CONST_D3DZBUFFERTYPE**

CONST_D3DBACKBUFFERTYPE

[#]Defines constants that describe the type of back buffer.

```
Enum CONST_D3DBACKBUFFERTYPE
   D3DBACKBUFFER_TYPE_MONO  = 0
   D3DBACKBUFFER_TYPE_LEFT  = 1
   D3DBACKBUFFER_TYPE_RIGHT = 2
End Enum
```

IDH_CONST_D3DBACKBUFFERTYPE_graphicsvb

Constants

D3DBACKBUFFER_TYPE_MONO
Specifies a non-stereo swap chain.

D3DBACKBUFFER_TYPE_LEFT
Specifies the left side of a stereo pair in a swap chain.

D3DBACKBUFFER_TYPE_RIGHT
Specifies the right side of a stereo pair in a swap chain.

Remarks

Note that stereo view in not supported in DirectX 8.0, so the D3DBACKBUFFER_TYPE_LEFT and D3DBACKBUFFER_TYPE_RIGHT values of this enumeration are not used by Direct3D.

See Also

Direct3DDevice8.GetBackBuffer, **Direct3DSwapChain8.GetBackBuffer**

CONST_D3DBLEND

#Defines the supported blend mode.

```
Enum CONST_D3DBLEND
   D3DBLEND_ZERO         = 1
   D3DBLEND_ONE          = 2
   D3DBLEND_SRCCOLOR     = 3
   D3DBLEND_INVSRCCOLOR  = 4
   D3DBLEND_SRCALPHA     = 5
   D3DBLEND_INVSRCALPHA  = 6
   D3DBLEND_DESTALPHA    = 7
   D3DBLEND_INVDESTALPHA = 8
   D3DBLEND_DESTCOLOR    = 9
   D3DBLEND_INVDESTCOLOR = 10
   D3DBLEND_SRCALPHASAT  = 11
   D3DBLEND_BOTHSRCALPHA = 12
   D3DBLEND_BOTHINVSRCALPHA = 13
End Enum
```

Constants

D3DBLEND_ZERO
Blend factor is (0, 0, 0, 0).

D3DBLEND_ONE

IDH_CONST_D3DBLEND_graphicsvb

Blend factor is (1, 1, 1, 1).

D3DBLEND_SRCCOLOR
Blend factor is (R_s, G_s, B_s, A_s).

D3DBLEND_INVSRCCOLOR
Blend factor is (1-R_s, 1-G_s, 1-B_s, 1-A_s).

D3DBLEND_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DBLEND_INVSRCALPHA
Blend factor is (1-A_s, 1-A_s, 1-A_s, 1-A_s).

D3DBLEND_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DBLEND_INVDESTALPHA
Blend factor is (1-A_d, 1-A_d, 1-A_d, 1-A_d).

D3DBLEND_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DBLEND_INVDESTCOLOR
Blend factor is (1-R_d, 1-G_d, 1-B_d, 1-A_d).

D3DBLEND_SRCALPHASAT
Blend factor is (f, f, f, 1); f = min(A_s, 1-A_d).

D3DBLEND_BOTHSRCALPHA
Not supported.

D3DBLEND_BOTHINVSRCALPHA
Source blend factor is (1-A_s, 1-A_s, 1-A_s, 1-A_s), and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden.

Remarks

In the member descriptions above, the RGBA values of the source and destination are indicated by the subscripts *s* and *d*.

These flags are used to set the value of the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states for the **CONST_D3DRENDERSTATETYPE** enumeration.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DBLENDOP

#Defines the supported blend operations.

Enum CONST_D3DBLENDOP

IDH_CONST_D3DBLENDOP_graphicsvb

```
    D3DBLENDOP_ADD        = 1
    D3DBLENDOP_SUBTRACT    = 2
    D3DBLENDOP_REVSUBTRACT = 3
    D3DBLENDOP_MIN        = 4
    D3DBLENDOP_MAX        = 5
End Enum
```

Constants

D3DBLENDOP_ADD
Adds the destination to the source.
Result = Source + Destination

D3DBLENDOP_SUBTRACT
Subtract the destination subtracted from the source.
Result = Source - Destination

D3DBLENDOP_REVSUBTRACT
Subtract the source from the destination.
Result = Destination - Source

D3DBLENDOP_MIN
Return the minimum of the source and destination.
Result = MIN(Source, Destination)

D3DBLENDOP_MAX
Return the maximum of the source and destination.
Result = MAX(Source, Destination)

Remarks

This enumerated type defines values used by the **D3DRS_BLENDOP** render state.

See Also

D3DCAPS8, **CONST_D3DRENDERSTATETYPE**

CONST_D3DCAPS2FLAGS

#Defines driver-specific capabilities.

```
Enum CONST_D3DCAPS2FLAGS
    D3DCAPS2_NO2DDURING3DSCENE =        2
    D3DCAPS2_FULLSCREENGAMMA   =   131072 (&H20000)
    D3DCAPS2_CANRENDERWINDOWED =   524288 (&H80000)
    D3DCAPS2_CANCALIBRATEGAMMA =  1048576 (&H100000)
```

IDH_CONST_D3DCAPS2FLAGS_graphicsvb

```
    D3DCAPS2_RESERVED        = 33554432 (&H2000000)
End Enum
```

Constants

D3DCAPS2_NO2DDURING3DSCENE

When the D3DCAPS2_NO2DDURING3DSCENE capability is set by the driver, it means that 2-D operations cannot be performed in between calls to **Direct3DDevice8.BeginScene** and **Direct3DDevice8.EndScene**.

Typically, this capability is set by hardware that partitions the scene and then renders each partition in sequence. The partitioning is performed in the driver, and the hardware will contain a small color and depth buffer that corresponds to the size of the image partition. On this type of rendering hardware, it is typical that once each part of the image is rendered, the data in the color buffers are written to video memory and the contents of the depth-buffer are discarded. Also, note that 3-D rendering does not start until **EndScene** is encountered. Then, the scene is processed in regions. As such, the processing order cannot be guaranteed. For example, the first region, typically the upper left corner of the window, that is processed may include the last triangle in the frame. To contrast this with more traditional graphics systems, in the typical system each command is processed sequentially in the order that it was sent. The 2-D operations are implied to occur at some fixed point in the processing. In the systems that set D3DCAPS2_NO2DDURING3DSCENE the processing order is not guaranteed, so 2-D operations that are encountered during 3-D rendering may be discarded by the display adapter.

In general, it is recommended that 2-D operations be performed outside of a **BeginScene** and **EndScene** pair. If 2-D operations are to be performed between a **BeginScene** and **EndScene** pair, then it is necessary to check the D3DCAPS2_NO2DDURING3DSCENE capability. If it is set, the application must expect that any 2-D operation that occurs between **BeginScene** and **EndScene** will be discarded. For more information on writing applications for systems that set D3DCAPS2_NO2DDURING3DSCENE, see Remarks.

D3DCAPS2_FULLSCREENGAMMA

The driver supports dynamic gamma ramp adjustment in full-screen mode.

D3DCAPS2_CANRENDERWINDOWED

The driver is capable of rendering in windowed mode.

D3DCAPS2_CANCALIBRATEGAMMA

The system has a calibrator installed that can automatically adjust the gamma ramp so that the result is identical on all systems that have a calibrator. To invoke the calibrator when setting new gamma levels, use the D3DSGR_CALIBRATE flag when calling the **Direct3DDevice8.SetGammaRamp** method. Calibrating gamma ramps incurs some processing overhead and should not be used frequently.

D3DCAPS2_RESERVED

This flag is reserved for future use. Do not use.

Remarks

These flags may be combined and present in the **Caps2** member of **D3DCAPS8**.

For systems that set the D3DCAPS2_NO2DDURING3DSCENE capability flag, performance problems may occur if you use a texture and then modify it during a scene. This is true on all hardware, but it is more severe on hardware that exposes the D3DCAPS2_NO2DDURING3DSCENE capability. If D3DCAPS2_NO2DDURING3DSCENE is present on the hardware, application-based texture management should ensure that no texture used in the current **BeginScene** and **EndScene** block is evicted unless absolutely necessary.

In the case of extremely high texture usage within a scene (that is, where you modify a texture that you have used in the scene and there is no spare texture memory available), the results are undefined. For such systems, the contents of the z buffer become invalid at **EndScene**. Applications should not call **Direct3DDevice8.CopyRects** to or from the back buffer on this type of hardware inside a **BeginScene** and **EndScene** pair. In addition, applications should not try to access the z-buffer if the D3DPRASTERCAPS_ZBUFFERLESSHSR capability flag is set. Finally, applications should not lock the back buffer or the z buffer inside a **BeginScene** and **EndScene** pair.

See Also

D3DCAPS8

CONST_D3DCAPSFLAGS

[#]Defines driver-specific capabilities.

```
Enum CONST_D3DCAPSFLAGS
   D3DCAPS_READ_SCANLINE = 131072 (&H20000)
End Enum
```

Constants

D3DCAPS_READ_SCANLINE
Display hardware is capable of returning the current scan line.

Remarks

This flag may be present in the **Caps** member of **D3DCAPS8**.

[#] IDH_CONST_D3DCAPSFLAGS_graphicsvb

See Also

D3DCAPS8

CONST_D3DCLEARFLAGS

#Defines constants that indicate which surface should be cleared.

```
Enum CONST_D3DCLEARFLAGS
   D3DCLEAR_TARGET  = 1
   D3DCLEAR_ZBUFFER = 2
   D3DCLEAR_STENCIL = 4
End Enum
```

Constants

D3DCLEAR_TARGET
Clear the render target.

D3DCLEAR_ZBUFFER
Clear the depth buffer.

D3DCLEAR_STENCIL
Clear the stencil buffer.

See Also

Direct3DDevice8.Clear

CONST_D3DCLIPFLAGS

#Defines the clip status.

```
Enum CONST_D3DCLIPFLAGS
   D3DCS_LEFT   =   1
   D3DCS_RIGHT  =   2
   D3DCS_TOP    =   4
   D3DCS_BOTTOM =   8
   D3DCS_FRONT  =  16 (&H10)
   D3DCS_BACK   =  32 (&H20)
   D3DCS_PLANE0 =  64 (&H40)
   D3DCS_PLANE1 = 128 (&H80)
   D3DCS_PLANE2 = 256 (&H100)
   D3DCS_PLANE3 = 512 (&H200)
   D3DCS_PLANE4 = 1024 (&H400)
```

IDH_CONST_D3DCLEARFLAGS_graphicsvb
IDH_CONST_D3DCLIPFLAGS_graphicsvb

```
    D3DCS_PLANE5 = 2048 (&H800)
    D3DCS_ALL    = 4095 (&HFFF)
End Enum
```

Constants

D3DCS_LEFT
All vertices are clipped by the left plane of the viewing frustum.

D3DCS_RIGHT
All vertices are clipped by the right plane of the viewing frustum.

D3DCS_TOP
All vertices are clipped by the top plane of the viewing frustum.

D3DCS_BOTTOM
All vertices are clipped by the bottom plane of the viewing frustum.

D3DCS_FRONT
All vertices are clipped by the front plane of the viewing frustum.

D3DCS_BACK
All vertices are clipped by the back plane of the viewing frustum.

D3DCS_PLANE0 through D3DCS_PLANE5
Application-defined clipping planes.

D3DCS_ALL
Combination of all clip flags.

Remarks

These flags may be combined and present in the **ClipUnion** and **ClipIntersection** member of **D3DCLIPSTATUS8**.

See Also

D3DCLIPSTATUS8

CONST_D3DCLIPPLANEFLAGS

#Defines constants that enable user-defined clipping planes.

```
Enum CONST_D3DCLIPPLANEFLAGS
    D3DCLIPPLANE0 =  1
    D3DCLIPPLANE1 =  2
    D3DCLIPPLANE2 =  4
    D3DCLIPPLANE3 =  8
    D3DCLIPPLANE4 = 16 (&H10)
    D3DCLIPPLANE5 = 32 (&H20)
```

IDH_CONST_D3DCLIPPLANEFLAGS_graphicsvb

```
End Enum
```

Constants

D3DCLIPPLANE0 through D3DCLIPPLANE5
Enable the clipping plane, or planes, at the corresponding index (0 through 5). These values can be combined to simultaneously enable multiple clipping planes.

Remarks

These flags are used to set the value of the D3DRS_CLIPPLANEENABLE render state for the **CONST_D3DRENDERSTATETYPE** enumeration.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DCMPFUNC

#Defines supported compare functions.

```
Enum CONST_D3DCMPFUNC
   D3DCMP_NEVER        = 1
   D3DCMP_LESS         = 2
   D3DCMP_EQUAL        = 3
   D3DCMP_LESSEQUAL    = 4
   D3DCMP_GREATER      = 5
   D3DCMP_NOTEQUAL     = 6
   D3DCMP_GREATEREQUAL = 7
   D3DCMP_ALWAYS       = 8
End Enum
```

Constants

D3DCMP_NEVER
Always fail the test.

D3DCMP_LESS
Accept the new pixel if its value is less than the value of the current pixel.

D3DCMP_EQUAL
Accept the new pixel if its value equals the value of the current pixel.

D3DCMP_LESSEQUAL
Accept the new pixel if its value is less than or equal to the value of the current pixel.

IDH_CONST_D3DCMPFUNC_graphicsvb

D3DCMP_GREATER
Accept the new pixel if its value is greater than the value of the current pixel.

D3DCMP_NOTEQUAL
Accept the new pixel if its value does not equal the value of the current pixel.

D3DCMP_GREATEREQUAL
Accept the new pixel if its value is greater than or equal to the value of the current pixel.

D3DCMP_ALWAYS
Always pass the test.

Remarks

These flags are used to set the value of the D3DRS_ZFUNC, D3DRS_ALPHAFUNC, and D3DRS_STENCILFUNC render states for the **CONST_D3DRENDERSTATETYPE** enumeration.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DCOLORWRITEENABLEFLAGS

#Defines color channels for updating during 3-D rendering.

```
Enum CONST_D3DCOLORWRITEENABLEFLAGS
   D3DCOLORWRITEENABLED_RED   = 1
   D3DCOLORWRITEENABLED_GREEN = 2
   D3DCOLORWRITEENABLED_BLUE  = 4
   D3DCOLORWRITEENABLED_ALPHA = 8
End Enum
```

Constants

D3DCOLORWRITEENABLED_RED
Specifies the red color channel.

D3DCOLORWRITEENABLED_GREEN
Specifies the green color channel.

D3DCOLORWRITEENABLED_BLUE
Specifies the blue color channel.

D3DCOLORWRITEENABLED_ALPHA
Specifies the alpha color channel.

IDH_CONST_D3DCOLORWRITEENABLEFLAGS_graphicsvb

Remarks

These flags are used to set the value of the D3DRS_COLORWRITEENABLE render state for the **CONST_D3DRENDERSTATETYPE** enumeration.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DCONST

#Defines miscellaneous constants used by Microsoft® Direct3D®.

```
Enum CONST_D3DCONST
   D3D_OK                          =       0
   D3DADAPTER_DEFAULT              =       0
   D3DCURSOR_IMMEDIATE_UPDATE      =       1
   D3DENUM_HOST_ADAPTER            =       1
   D3DPRESENTFLAG_LOCKABLE_BACKBUFFER =    1
   D3DPV_DONOTCOPYDATA             =       1
   D3DENUM_NO_WHQL_LEVEL           =       2
   D3DPRESENT_BACK_BUFFERS_MAX     =       3
   VALID_D3DENUM_FLAGS             =       3
   D3DMAXNUMPRIMITIVES             =   65535 (&HFFFF)
   D3DMAXNUMVERTICES               =   65535 (&HFFFF)
   D3DCURRENT_DISPLAY_MODE         = 15728639 (&HEFFFFF)
End Enum
```

Constants

D3D_OK
: Success code for compatibility with some methods that return error codes rather than setting **Err.Number**.

D3DADAPTER_DEFAULT
: The primary display adapter.

D3DCURSOR_IMMEDIATE_UPDATE
: Update cursor at the refresh rate.

D3DENUM_HOST_ADAPTER
: Not documented for this release.

D3DPRESENTFLAG_LOCKABLE_BACKBUFFER
: Set this flag if the application requires the ability to lock the back-buffer directly. Note that back buffers are not lockable unless the application specifies D3DPRESENTFLAG_LOCKABLE_BACKBUFFER at

IDH_CONST_D3DCONST_graphicsvb

Direct3D8.CreateDevice and **Direct3DDevice8.Reset** time. Lockable back buffers incur a performance cost on some graphics hardware configurations.

Performing a lock operation (or using **Direct3DDevice8.CopyRects** to read/write) on the lockable back-buffer will decrease performance on many cards. In this case, you should consider using textured triangles to move data to the back buffer.

D3DPV_DONOTCOPYDATA
Prevents the system from copying vertex data not affected by the vertex operation into the destination buffer.

D3DENUM_NO_WHQL_LEVEL
Forces the **WHQLLevel** member of the **D3DADAPTER_IDENTIFIER8** type to be zero, meaning that no information is returned for the WHQL certification date. Setting this flag will avoid the one or two second time penalty incurred to determine the WHQL certification date.

D3DPRESENT_BACK_BUFFERS_MAX
Maximum number of back-buffers supported in Microsoft® DirectX® 8.0.

VALID_D3DENUM_FLAGS
Not documented for this release.

D3DMAXNUMPRIMITIVES
Maximum number of primitives a user can pass to any Direct3D drawing method.

D3DMAXNUMVERTICES
Maximum number of vertices user can pass to any Direct3D drawing function or vertex buffer creation method.

D3DCURRENT_DISPLAY_MODE
Denotes the current desktop display mode on an adapter.

See Also

Direct3DDevice8.SetCursorPosition

CONST_D3DCREATEFLAGS

#Defines Microsoft® Direct3D® device creation flags.

```
Enum CONST_D3DCREATEFLAGS
   D3DCREATE_FPU_PRESERVE           = 2
   D3DCREATE_MULTITHREADED          = 4
   D3DCREATE_PUREDEVICE             = 16 (&H10)
   D3DCREATE_SOFTWARE_VERTEXPROCESSING = 32 (&H20)
   D3DCREATE_HARDWARE_VERTEXPROCESSING = 64 (&H40)
   D3DCREATE_MIXED_VERTEXPROCESSING    = 128 (&H80)
End Enum
```

IDH_CONST_D3DCREATEFLAGS_graphicsvb

Constants

D3DCREATE_FPU_PRESERVE
Indicates that the application needs either double precision FPU or FPU exceptions enabled. Direct3D sets the FPU state each time it is called. Setting the flag will reduce Direct3D performance.

D3DCREATE_MULTITHREADED
Indicates that the application requests Direct3D to be multithread safe. This makes Direct3D take its global critical section more frequently, which can degrade performance.

D3DCREATE_PUREDEVICE
Specifies hardware rasterization, transform, lighting, and shading. This flag should be specified only when D3DCREATE_HARDWARE_VERTEXPROCESSING is also specified.

D3DCREATE_SOFTWARE_VERTEXPROCESSING
Specifies software vertex processing. See Remarks.

D3DCREATE_HARDWARE_VERTEXPROCESSING
Specifies hardware vertex processing. See Remarks.

D3DCREATE_MIXED_VERTEXPROCESSING
Specifies mixed (both software and hardware) vertex processing. See Remarks.

Remarks

Note that D3DCREATE_HARDWARE_VERTEXPROCESSING, D3DCREATE_MIXED_VERTEXPROCESSING, and D3DCREATE_SOFTWARE_VERTEXPROCESSING are mutually exclusive flags, and that at least one of these vertex processing flags must be specified when calling **Direct3D8.CreateDevice**.

See Also

Direct3D8.CreateDevice

CONST_D3DCUBEMAP_FACES

#Defines the faces of a cubemap.

```
Enum CONST_D3DCUBEMAP_FACES
    D3DCUBEMAP_FACE_POSITIVE_X = 0
    D3DCUBEMAP_FACE_NEGATIVE_X = 1
    D3DCUBEMAP_FACE_POSITIVE_Y = 2
    D3DCUBEMAP_FACE_NEGATIVE_Y = 3
    D3DCUBEMAP_FACE_POSITIVE_Z = 4
```

IDH_CONST_D3DCUBEMAP_FACES_graphicsvb

```
   D3DCUBEMAP_FACE_NEGATIVE_Z = 5
End Enum
```

Constants

D3DCUBEMAP_FACE_POSITIVE_X
Positive x-face of the cubemap.

D3DCUBEMAP_FACE_NEGATIVE_X
Negative x-face of the cubemap.

D3DCUBEMAP_FACE_POSITIVE_Y
Positive y-face of the cubemap.

D3DCUBEMAP_FACE_NEGATIVE_Y
Negative y-face of the cubemap.

D3DCUBEMAP_FACE_POSITIVE_Z
Positive z-face of the cubemap.

D3DCUBEMAP_FACE_NEGATIVE_Z
Negative z-face of the cubemap.

See Also

Direct3DCubeTexture8.AddDirtyRect, **Direct3DCubeTexture8.GetCubeMapSurface**, **Direct3DCubeTexture8.LockRect**, **Direct3DCubeTexture8.UnlockRect**

CONST_D3DCULL

#Defines the supported culling modes.

```
Enum CONST_D3DCULL
   D3DCULL_NONE = 1
   D3DCULL_CW   = 2
   D3DCULL_CCW  = 3
End Enum
```

Constants

D3DCULL_NONE
Do not cull back faces.

D3DCULL_CW
Cull back faces with clockwise vertices.

D3DCULL_CCW
Cull back faces with counterclockwise vertices.

IDH_CONST_D3DCULL_graphicsvb

Remarks

The values in this enumerated type are used to set the value of the D3DRS_CULLMODE render state for the **CONST_D3DRENDERSTATETYPE** enumeration.

The culling modes define how back faces are culled when rendering a geometry.

See Also

D3DCAPS8, **CONST_D3DRENDERSTATETYPE**

CONST_D3DCURSORCAPSFLAGS

#Defines cursor capabilities.

```
Enum CONST_D3DCURSORCAPSFLAGS
   D3DCURSORCAPS_COLOR   = 1
   D3DCURSORCAPS_LOWRES  = 2
End Enum
```

Constants

D3DCURSORCAPS_COLOR
A two-color black-and-white cursor is supported in hardware.

D3DCURSORCAPS_LOWRES
A full-color cursor is supported in hardware.

Remarks

Microsoft® Direct3D® for Microsoft DirectX® 8.0 does not define alpha-blending cursor capabilities.

See Also

D3DCAPS8

CONST_D3DDEBUGMONITORTOKENS

#Defines the debug monitor tokens.

```
Enum CONST_D3DDEBUGMONITORTOKENS
```

IDH_CONST_D3DCURSORCAPSFLAGS_graphicsvb
IDH_CONST_D3DDEBUGMONITORTOKENS_graphicsvb

```
    D3DDMT_ENABLE  = 0
    D3DDMT_DISABLE = 1
End Enum
```

Constants

D3DDMT_ENABLE
Enable the debug monitor.

D3DDMT_DISABLE
Disable the debug monitor.

Remarks

The values in this enumerated type are used by the D3DRS_DEBUGMONITORTOKEN render state and are relevant only for debug builds.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DDEVCAPSFLAGS

#Defines the capabilities of the device.

```
Enum CONST_D3DDEVCAPSFLAGS
    D3DDEVCAPS_EXECUTESYSTEMMEMORY     =       16 (&H10)
    D3DDEVCAPS_EXECUTEVIDEOMEMORY      =       32 (&H20)
    D3DDEVCAPS_TLVERTEXSYSTEMMEMORY    =       64 (&H40)
    D3DDEVCAPS_TLVERTEXVIDEOMEMORY     =      128 (&H80)
    D3DDEVCAPS_TEXTURESYSTEMMEMORY     =      256 (&H100)
    D3DDEVCAPS_TEXTUREVIDEOMEMORY      =      512 (&H200)
    D3DDEVCAPS_DRAWPRIMITIVES2         =     8192 (&H2000)
    D3DDEVCAPS_DRAWPRIMTLVERTEX        =     1024 (&H400)
    D3DDEVCAPS_CANRENDERAFTERFLIP      =     2048 (&H800)
    D3DDEVCAPS_TEXTURENONLOCALVIDMEM   =     4096 (&H1000)
    D3DDEVCAPS_SEPARATETEXTUREMEMORIES =    16384 (&H4000)
    D3DDEVCAPS_DRAWPRIMITIVES2EX       =    32768 (&H8000)
    D3DDEVCAPS_HWTRANSFORMANDLIGHT     =    65536 (&H10000)
    D3DDEVCAPS_CANBLTSYSTONONLOCAL     =   131072 (&H20000)
    D3DDEVCAPS_HWRASTERIZATION         =   524288 (&H80000)
    D3DDEVCAPS_PUREDEVICE              =  1048576 (&H100000)
    D3DDEVCAPS_QUINTICRTPATCHES        =  2097152 (&H200000)
    D3DDEVCAPS_RTPATCHES               =  4194304 (&H400000)
```

IDH_CONST_D3DDEVCAPSFLAGS_graphicsvb

```
    D3DDEVCAPS_RTPATCHHANDLEZERO      =  8388608 (&H800000)
    D3DDEVCAPS_NPATCHES               = 16777216 (&H1000000)
End Enum
```

Constants

D3DDEVCAPS_EXECUTESYSTEMMEMORY
Device can use execute buffers from system memory.

D3DDEVCAPS_EXECUTEVIDEOMEMORY
Device can use execute buffers from video memory.

D3DDEVCAPS_TLVERTEXSYSTEMMEMORY
Device can use buffers from system memory for transformed and lit vertices.

D3DDEVCAPS_TLVERTEXVIDEOMEMORY
Device can use buffers from video memory for transformed and lit vertices.

D3DDEVCAPS_TEXTURESYSTEMMEMORY
Device can retrieve textures from system memory.

D3DDEVCAPS_TEXTUREVIDEOMEMORY
Device can retrieve textures from device memory.

D3DDEVCAPS_DRAWPRIMITIVES2
Not documented for this release.

D3DDEVCAPS_DRAWPRIMTLVERTEX
Device exports a DrawPrimitive-aware hardware abstraction layer (HAL).

D3DDEVCAPS_CANRENDERAFTERFLIP
Device can queue rendering commands after a page flip. Applications do not change their behavior if this flag is set; this capability simply means that the device is relatively fast.

D3DDEVCAPS_TEXTURENONLOCALVIDMEM
Device can retrieve textures from non-local video memory.

D3DDEVCAPS_SEPARATETEXTUREMEMORIES
Device is texturing from separate memory pools.

D3DDEVCAPS_DRAWPRIMITIVES2EX
Not documented for this release.

D3DDEVCAPS_HWTRANSFORMANDLIGHT
Device can support transformation and lighting in hardware.

D3DDEVCAPS_CANBLTSYSTONONLOCAL
Device supports blits from system-memory textures to nonlocal video-memory textures.

D3DDEVCAPS_HWRASTERIZATION
Device has hardware acceleration for scene rasterization.

D3DDEVCAPS_PUREDEVICE
Device can support rasterization, transform, lighting, and shading in hardware.

D3DDEVCAPS_QUINTICRTPATCHES
Device supports quintic béziers and B-splines.

D3DDEVCAPS_RTPATCHES
Device supports high-order surfaces.

D3DDEVCAPS_RTPATCHHANDLEZERO
Indicates that high-order surfaces can be drawn efficiently using a handle value of 0.

D3DDEVCAPS_NPATCHES
Device supports N patches.

Remarks

These flags may be combined and present in the **DevCaps** member of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DDEVTYPE

[#]Defines device types.

```
Enum CONST_D3DDEVTYPE
    D3DDEVTYPE_HAL      = 1
    D3DDEVTYPE_REF      = 2
    D3DDEVTYPE_SW       = 3
End Enum
```

Constants

D3DDEVTYPE_HAL
Hardware rasterization and shading with software, hardware, or mixed transform and lighting.

D3DDEVTYPE_REF
Microsoft® Direct3D® features are implemented in software.

D3DDEVTYPE_SW
A pluggable software device that has been registered with Direct3D using **Direct3D8.RegisterSoftwareDevice**.

See Also

Direct3D8.CheckDeviceFormat, **Direct3D8.CheckDeviceMultiSampleType**, **Direct3D8.CheckDeviceType**, **Direct3D8.CreateDevice**, **Direct3D8.GetDeviceCaps**, **D3DDEVICE_CREATION_PARAMETERS**

[#] IDH_CONST_D3DDEVTYPE_graphicsvb

CONST_D3DERR

#Defines error codes raised by the system. For descriptions of these error codes, see Error Codes.

CONST_D3DFILLMODE

#Defines constants that describe the fill mode.

```
Enum CONST_D3DFILLMODE
   D3DFILL_POINT     = 1
   D3DFILL_WIREFRAME = 2
   D3DFILL_SOLID     = 3
End Enum
```

Constants

D3DFILL_POINT
Fill points.

D3DFILL_WIREFRAME
Fill wireframes. This fill mode currently does not work for clipped primitives when you use the DrawPrimitive methods.

D3DFILL_SOLID
Fill solids.

Remarks

These flags are used to set the value of the D3DRS_FILLMODE render state for the **CONST_D3DRENDERSTATETYPE** enumeration.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DFOGMODE

#Defines constants that describe the fog mode.

```
Enum CONST_D3DFOGMODE
   D3DFOG_NONE   = 0
   D3DFOG_EXP    = 1
   D3DFOG_EXP2   = 2
```

IDH_CONST_D3DERR_graphicsvb
IDH_CONST_D3DFILLMODE_graphicsvb
IDH_CONST_D3DFOGMODE_graphicsvb

```
    D3DFOG_LINEAR = 3
End Enum
```

Constants

D3DFOG_NONE
No fog effect.

D3DFOG_EXP
Fog effect intensifies exponentially, according to the following formula.

$$f = \frac{1}{e^{d \times density}}$$

D3DFOG_EXP2
Fog effect intensifies exponentially with the square of the distance, according to the following formula.

$$f = \frac{1}{e^{(d \times density)^2}}$$

D3DFOG_LINEAR
Fog effect intensifies linearly between the start and end points, according to the following formula.

$$f = \frac{}{end - start}$$

This is the only fog mode currently supported.

Remarks

These flags are used to set the value of the D3DRS_FOGTABLEMODE and D3DRS_FOGVERTEXMODE render states for the **CONST_D3DRENDERSTATETYPE** enumeration.

Fog can be considered a measure of visibility—the lower the fog value produced by a fog equation, the less visible an object is.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DFORMAT

#Defines the various types of surface formats.

```
Enum CONST_D3DFORMAT
    D3DFMT_UNKNOWN      =   0
    D3DFMT_R8G8B8       =  20 (&H14)
    D3DFMT_A8R8G8B8     =  21 (&H15)
    D3DFMT_X8R8G8B8     =  22 (&H16)
    D3DFMT_R5G6B5       =  23 (&H17)
    D3DFMT_X1R5G5B5     =  24 (&H18)
    D3DFMT_A1R5G5B5     =  25 (&H19)
    D3DFMT_A4R4G4B4     =  26 (&H1A)
    D3DFMT_R3G3B2       =  27 (&H1B)
    D3DFMT_A8           =  28 (&H1C)
    D3DFMT_A8R3G3B2     =  29 (&H1D)
    D3DFMT_X4R4G4B4     =  30 (&H1E)
    D3DFMT_A8P8         =  40 (&H28)
    D3DFMT_P8           =  41 (&H29)
    D3DFMT_L8           =  50 (&H32)
    D3DFMT_A8L8         =  51 (&H33)
    D3DFMT_A4L4         =  52 (&H34)
    D3DFMT_V8U8         =  60 (&H3C)
    D3DFMT_L6V5U5       =  61 (&H3D)
    D3DFMT_X8L8V8U8     =  62 (&H3E)
    D3DFMT_Q8W8V8U8     =  63 (&H3F)
    D3DFMT_V16U16       =  64 (&H40)
    D3DFMT_W11V11U10    =  65 (&H41)
    D3DFMT_UYVY         = 1498831189 (&H59565955)
    D3DFMT_YUY2         =  844715353 (&H32595559)
    D3DFMT_DXT1         =  827611204 (&H31545844)
    D3DFMT_DXT2         =  844388420 (&H32545844)
    D3DFMT_DXT3         =  861165636 (&H33545844)
    D3DFMT_DXT4         =  877942852 (&H34545844)
    D3DFMT_DXT5         =  894720068 (&H35545844)
    D3DFMT_D16_LOCKABLE =  70 (&H46)
    D3DFMT_D32          =  71 (&H47)
    D3DFMT_D15S1        =  73 (&H49)
    D3DFMT_D24S8        =  75 (&H4B)
    D3DFMT_D16          =  80 (&H50)
    D3DFMT_D24X8        =  77 (&H4D)
    D3DFMT_D24X4S4      =  79 (&H4F)
    D3DFMT_VERTEXDATA   = 100 (&H64)
    D3DFMT_INDEX16      = 101 (&H65)
```

IDH_CONST_D3DFORMAT_graphicsvb

```
    D3DFMT_INDEX32      = 102 (&H66)
End Enum
```

Constants

D3DFMT_UNKNOWN
Surface format is unknown.

D3DFMT_R8G8B8
24-bit RGB pixel format.

D3DFMT_A8R8G8B8
32-bit ARGB pixel format with alpha.

D3DFMT_X8R8G8B8
32-bit RGB pixel format where 8 bits are reserved for each color.

D3DFMT_R5G6B5
16-bit RGB pixel format.

D3DFMT_X1R5G5B5
16-bit pixel format where 5 bits are reserved for each color.

D3DFMT_A1R5G5B5
16-bit pixel format where 5 bits are reserved for each color and 1 bit is reserved for alpha (transparent texel).

D3DFMT_A4R4G4B4
16-bit ARGB pixel format.

D3DFMT_R3G3B2
8-bit RGB texture format.

D3DFMT_A8
8-bit alpha only.

D3DFMT_A8R3G3B2
16-bit ARGB texture format.

D3DFMT_X4R4G4B4
16-bit RGB pixel format where 4 bits are reserved for each color.

D3DFMT_A8P8
Surface is 8-bit color indexed with 8 bits of alpha.

D3DFMT_P8
Surface is 8-bit color indexed.

D3DFMT_L8
8-bit luminance only.

D3DFMT_A8L8
16-bit alpha luminance.

D3DFMT_A4L4
8-bit alpha luminance.

D3DFMT_V8U8
16-bit bump-map format.

D3DFMT_L6V5U5
16-bit bump-map format with luminance.

D3DFMT_X8L8V8U8
32-bit bump-map format with luminance where 8 bits are reserved for each element.

D3DFMT_Q8W8V8U8
32-bit bump-map format.

D3DFMT_V16U16
32-bit bump-map format.

D3DFMT_W11V11U10
32-bit bump-map format.

D3DFMT_UYVY
UYVY format (PC98 compliance).

D3DFMT_YUY2
YUY2 format (PC98 compliance).

D3DFMT_DXT1
DXT1 compression texture format.

D3DFMT_DXT2
DXT2 compression texture format.

D3DFMT_DXT3
DXT3 compression texture format.

D3DFMT_DXT4
DXT4 compression texture format.

D3DFMT_DXT5
DXT5 compression texture format.

D3DFMT_D16_LOCKABLE
16-bit z-buffer bit depth. This is an application-lockable surface format.

D3DFMT_D32
32-bit z-buffer bit depth.

D3DFMT_D15S1
16-bit z-buffer bit depth where 15 bits are reserved for the depth channel and 1 bit is reserved for the stencil channel.

D3DFMT_D24S8
32-bit z-buffer bit depth where 24 bits are reserved for the depth channel and 8 bits are reserved for the stencil channel.

D3DFMT_D16
16-bit z-buffer bit depth.

D3DFMT_D24X8
32-bit z-buffer bit depth where 24 bits are reserved for the depth channel.

D3DFMT_D24X4S4
32-bit z-buffer bit depth where 24 bits are reserved for the depth channel and 4 bits are reserved for the stencil channel.

D3DFMT_VERTEXDATA

Describes a vertex buffer surface.

D3DFMT_INDEX16
16-bit index buffer bit depth.

D3DFMT_INDEX32
32-bit index buffer bit depth.

Remarks

Note that render target formats are restricted to D3DFMT_X1R5G5B5, D3DFMT_R5G6B5, D3DFMT_X8R8G8B8, and D3DFMT_A8R8G8B8.

The order of the bits is from the most significant bit (MSB) first, so D3DFMT_A8L8 indicates that the high byte of this two-byte format is alpha.

All depth-stencil formats except D3DFMT_D16_LOCKABLE indicate no particular bit ordering per pixel. They are not application-lockable, and the driver is allowed to consume more than the indicated number of bits per depth channel (but not stencil channel).

Pixel formats are denoted by opaque **Long** identifiers. The format of these **Long**s has been chosen to enable the expression of IHV-defined extension formats, and also to include the well-established FOURCC method. The set of formats understood by the Microsoft® Direct3D® run time is defined by **D3DFORMAT**.

Note that IHV-supplied formats and many FOURCC codes are not listed in the **D3DFORMAT** enumeration. The formats in this enumeration are unique in that they are sanctioned by the run time, meaning that the reference rasterizer will operate on all these types. The IHV-supplied formats will be supported by the individual IHVs on a card-by-card basis.

See Also

Direct3D8.CheckDeviceFormat, Four-Character Codes (FOURCC)

CONST_D3DFVFCAPSFLAGS

[#]Defines flexible vertex format capabilities.

```
Enum CONST_D3DFVFCAPSFLAGS
   D3DFVFCAPS_TEXCOORDCOUNTMASK  =   65535 (&HFFFF)
   D3DFVFCAPS_DONOTSTRIPELEMENTS =  524288 (&H80000)
   D3DFVFCAPS_PSIZE              = 1048576 (&H100000)
End Enum
```

IDH_CONST_D3DFVFCAPSFLAGS_graphicsvb

Constants

D3DFVFCAPS_TEXCOORDCOUNTMASK
: Masks the low 16 bits of the **FVFCaps** member of **D3DCAPS8**. Set the result to a variable of type Integer to calculate the total number of texture coordinate sets that the device can simultaneously use for multiple texture blending. (You can use up to eight texture coordinate sets for any vertex, but the device can blend only by using the specified number of texture coordinate sets.)

D3DFVFCAPS_DONOTSTRIPELEMENTS
: It is preferable that vertex elements not be stripped. That is, if the vertex format contains elements that are not used with the current render states, there is no need to regenerate the vertices. If this capability flag is not present, stripping extraneous elements from the vertex format provides better performance.

D3DFVFCAPS_PSIZE
: The absence of D3DFVFCAPS_PSIZE indicates that the device does not support D3DFVF_PSIZE for pre-transformed vertices. In this case the base point size always comes from D3DRS_POINTSIZE. This capability applies to fixed-function vertex processing in software only. D3DFVF_PSIZE is always supported when doing software vertex processing.

  The output point size written by a vertex shader is always supported, and for vertex shaders any input can contribute to the output point size.

  D3DFVF_PSIZE is always supported for post-transformed vertices. For more information on D3DFVF_PSIZE, see flexible vertex format flags.

Remarks

These flags may be combined and present in the **FVFCaps** member of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DFVFFLAGS

#Defines flexible vertex format flags.

For details, see flexible vertex format flags.

CONST_D3DFVFTEXTUREFORMATS

IDH_CONST_D3DFVFFLAGS_graphicsvb

#Defines bit patterns that are used to identify texture coordinate formats within a flexible vertex format description. The members of this enumeration can be combined within a flexible vertex format description by using the **OR** operator.

```
Enum CONST_D3DFVFTEXTUREFORMATS
    D3DFVF_TEXTUREFORMAT1  =       3
    D3DFVF_TEXTUREFORMAT2  =       0
    D3DFVF_TEXTUREFORMAT3  =       1
    D3DFVF_TEXTUREFORMAT4  =       2
    D3DFVF_TEXCOORDSIZE1_0 =   196608 (&H30000)
    D3DFVF_TEXCOORDSIZE1_1 =   786432 (&HC0000)
    D3DFVF_TEXCOORDSIZE1_2 =  3145728 (&H300000)
    D3DFVF_TEXCOORDSIZE1_3 = 12582912 (&HC00000)
    D3DFVF_TEXCOORDSIZE2_0 =       0
    D3DFVF_TEXCOORDSIZE2_1 =       0
    D3DFVF_TEXCOORDSIZE2_2 =       0
    D3DFVF_TEXCOORDSIZE2_3 =       0
    D3DFVF_TEXCOORDSIZE3_0 =    65536 (&H10000)
    D3DFVF_TEXCOORDSIZE3_1 =   262144 (&H40000)
    D3DFVF_TEXCOORDSIZE3_2 =  1048576 (&H100000)
    D3DFVF_TEXCOORDSIZE3_3 =  4194304 (&H400000)
    D3DFVF_TEXCOORDSIZE4_0 =   131072 (&H20000)
    D3DFVF_TEXCOORDSIZE4_1 =   524288 (&H80000)
    D3DFVF_TEXCOORDSIZE4_2 =  2097152 (&H200000)
    D3DFVF_TEXCOORDSIZE4_3 =  8388608 (&H800000)
End Enum
```

Constants

D3DFVF_TEXTUREFORMAT1 to D3DFVF_TEXTUREFORMAT4
: Values that can be used to calculate 1-, 2-, 3-, or 4-dimensional bit masks for identifying texture coordinate formats. See Remarks.

D3DFVF_TEXCOORDSIZE1_0 to D3DFVF_TEXCOORDSIZE4_3
: These bit patterns specify texture coordinate formats of different sizes on specific vertices. TEXCOORDSIZE*m_n* specifies that *m* coordinates are to be associated with the *(n+1)*th position in the vertex format.

Remarks

D3DFVF_TEXTUREFORMAT1 to D3DFVF_TEXTUREFORMAT4 specify the number of values that define a texture coordinate set. The D3DFVF_TEXTUREFORMAT1 indicates one-dimensional texture coordinates, D3DFVF_TEXTUREFORMAT2 indicates two-dimensional texture coordinates, and

---

IDH_CONST_D3DFVFTEXTUREFORMATS_graphicsvb

so on. The following sample functions show you how to create bit patterns to describe the number of elements used by a particular set of texture coordinates.

```
'Equivalent to D3DFVF_TEXCOORDSIZE1_coordIndex, specifying that one
'texture coordinate is used for the vertex format specified by coordIndex
Function D3DFVF_TEXCOORDSIZE1(coordIndex As Long) As Long
   D3DFVF_TEXCOORDSIZE1 = D3DFVF_TEXTUREFORMAT1 * 2 ^ ((coordIndex * 2) + 16)
End Function

'Equivalent to D3DFVF_TEXCOORDSIZE1_coordIndex, specifying that two
'texture coordinates are used for the vertex format specified by coordIndex
Function D3DFVF_TEXCOORDSIZE2(coordIndex As Long) As Long
   D3DFVF_TEXCOORDSIZE2 = D3DFVF_TEXTUREFORMAT2
End Function

'Equivalent to D3DFVF_TEXCOORDSIZE1_coordIndex, specifying that three
'texture coordinates are used for the vertex format specified by coordIndex
Function D3DFVF_TEXCOORDSIZE3(coordIndex As Long) As Long
   D3DFVF_TEXCOORDSIZE3 = D3DFVF_TEXTUREFORMAT3 * 2 ^ ((coordIndex * 2) + 16)
End Function

'Equivalent to D3DFVF_TEXCOORDSIZE1_coordIndex, specifying that four
'texture coordinate are used for the vertex format specified by coordIndex
Function D3DFVF_TEXCOORDSIZE4(coordIndex As Long) As Long
   D3DFVF_TEXCOORDSIZE4 = D3DFVF_TEXTUREFORMAT4 * 2 ^ ((coordIndex * 2) + 16)
End Function
```

See Also

Flexible Vertex Format Flags

CONST_D3DBASISTYPE

#Defines the basis type of a high-order surface.

```
Enum CONST_D3DBASISTYPE
   D3DBASIS_BEZIER     = 0
   D3DBASIS_BSPLINE    = 1
   D3DBASIS_INTERPOLATE = 2
End Enum
```

Constants

D3DBASIS_BEZIER

IDH_CONST_D3DBASISTYPE_graphicsvb

Input vertices are treated as a series of bézier patches. The number of vertices specified must be divisible by 3 + 1. Portions of the mesh beyond this criterion will not be rendered. Full continuity is assumed between sub-patches in the interior of the surface rendered by each call. Only the vertices at the corners of each sub-patch are guaranteed to lie on the resulting surface.

D3DBASIS_BSPLINE

Input vertices are treated as control points of a B-spline surface. The number of apertures rendered is 2 less than the number of apertures in that direction. In general, the generated surface does not contain the control vertices specified.

D3DBASIS_INTERPOLATE

An interpolating basis defines the surface so that the surface goes through all the input vertices specified.

Remarks

The members of **D3DBASISTYPE** specify the formulation to be used in evaluating the high-order surface primitive during tessellation.

See Also

D3DRECTPATCH_INFO, **D3DTRIPATCH_INFO**

CONST_D3DLIGHTTYPE

#Defines the light type.

```
Enum CONST_D3DLIGHTTYPE
   D3DLIGHT_POINT       = 1
   D3DLIGHT_SPOT        = 2
   D3DLIGHT_DIRECTIONAL = 3
End Enum
```

Constants

D3DLIGHT_POINT

Light is a point source. The light has a position in space and radiates light in all directions.

D3DLIGHT_SPOT

Light is a spotlight source. This light is similar a point light, except that the illumination is limited to a cone. This light type has a direction and several other parameters that determine the shape of the cone it produces. For information about these parameters, see the **D3DLIGHT8** type.

D3DLIGHT_DIRECTIONAL

IDH_CONST_D3DLIGHTTYPE_graphicsvb

Light is a directional source. This is equivalent to using a point light source at an infinite distance.

Remarks

Directional lights are slightly faster than point light sources, but point lights look a little better. Spotlights offer interesting visual effects but are computationally expensive.

See Also

D3DLIGHT8

CONST_D3DLINECAPS

#Defines locking flags.

```
Enum CONST_D3DLINECAPS
   D3DLINECAPS_TEXTURE  =  1
   D3DLINECAPS_ZTEST    =  2
   D3DLINECAPS_BLEND=   =  4
   D3DLINECAPS_ALPHACMP =  8
   D3DLINECAPS_FOG      = 16 (&H10)
End Enum
```

Constants

D3DLINECAPS_TEXTURE
Supports texture-mapping.

D3DLINECAPS_ZTEST
Supports z-buffer comparisons.

D3DLINECAPS_BLEND
Supports source-blending.

D3DLINECAPS_ALPHACMP
Supports alpha-test comparisons.

D3DLINECAPS_FOG
Supports fog.

CONST_D3DLOCKFLAGS

#Defines locking flags.

IDH_CONST_D3DLINECAPS_graphicsvb
IDH_CONST_D3DLOCKFLAGS_graphicsvb

```
Enum CONST_D3DLOCKFLAGS
   D3DLOCK_READONLY       =    16 (&H10)
   D3DLOCK_NOSYSLOCK      =  2048 (&H800)
   D3DLOCK_NOOVERWRITE    =  4096 (&H1000)
   D3DLOCK_DISCARD        =  8192 (&H2000)
   D3DLOCK_NO_DIRTY_UPDATE = 32768 (&H8000)
End Enum
```

Constants

D3DLOCK_READONLY

The application will not write to the buffer. This enables some optimizations. D3DLOCK_READONLY cannot be specified with D3DLOCK_DISCARD; nor can it be specified on an index or vertex buffer created with D3DUSAGE_WRITEONLY.

D3DLOCK_NOSYSLOCK

The default behavior of a video memory lock is to reserve a system-wide critical section, guaranteeing that no display mode changes will occur for the duration of the lock. This flag causes the system-wide critical section not to be held for the duration of the lock.

The lock operation is slightly more expensive, but can enable the system to perform other duties, such as moving the mouse cursor. This flag is useful for long duration locks, such as the lock of the back buffer for software rendering that would otherwise adversely affect system

D3DLOCK_NOOVERWRITE

Indicates that no indices or vertices that were referred to in drawing calls since the start of the frame or the last lock without this flag will be modified during the lock. This can enable optimizations only when the application is appending data to the index or vertex buffer. This flag is valid only on buffers created with D3DUSAGE_DYNAMIC.

D3DLOCK_DISCARD

The application overwrites, with a write-only operation, the entire buffer. This enables Direct3D to return a pointer to a new memory area so that dynamic memory access (DMA) and rendering from the old area do not stall. This flag is valid only for index and vertex buffers. This flag is valid only on buffers created with D3DUSAGE_DYNAMIC.

D3DLOCK_NO_DIRTY_UPDATE

By default, a lock on a resource adds a dirty region to that resource. This flag prevents any changes to the dirty state of the resource. Applications should use this flag when they have additional information about the actual set of regions changed during the lock operation. Note that this flag is not valid for vertex and index buffers.

CONST_D3DMATERIALCOLORSOURCE

#Defines the location at which a color or color component must be accessed for lighting calculations.

```
Enum CONST_D3DMATERIALCOLORSOURCE
   D3DMCS_MATERIAL = 0
   D3DMCS_COLOR1   = 1
   D3DMCS_COLOR2   = 2
End Enum
```

Constants

D3DMCS_MATERIAL
: Use the color from the current material.

D3DMCS_COLOR1
: Use the diffuse vertex color.

D3DMCS_COLOR2
: Use the specular vertex color.

Remarks

These flags are used to set the value of the following render states for the **CONST_D3DRENDERSTATETYPE** enumeration.

- D3DRS_DIFFUSEMATERIALSOURCE
- D3DRS_SPECULARMATERIALSOURCE
- D3DRS_AMBIENTMATERIALSOURCE
- D3DRS_EMISSIVEMATERIALSOURCE

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DMULTISAMPLE_TYPE

#Defines levels of full-scene multisampling that the device can apply.

```
Enum CONST_D3DMULTISAMPLE_TYPE
   D3DMULTISAMPLE_NONE      = 0
   D3DMULTISAMPLE_2_SAMPLES = 2
```

IDH_CONST_D3DMATERIALCOLORSOURCE_graphicsvb
IDH_CONST_D3DMULTISAMPLE_TYPE_graphicsvb

```
    D3DMULTISAMPLE_3_SAMPLES  =  3
    D3DMULTISAMPLE_4_SAMPLES  =  4
    D3DMULTISAMPLE_5_SAMPLES  =  5
    D3DMULTISAMPLE_6_SAMPLES  =  6
    D3DMULTISAMPLE_7_SAMPLES  =  7
    D3DMULTISAMPLE_8_SAMPLES  =  8
    D3DMULTISAMPLE_9_SAMPLES  =  9
    D3DMULTISAMPLE_10_SAMPLES = 10
    D3DMULTISAMPLE_11_SAMPLES = 11
    D3DMULTISAMPLE_12_SAMPLES = 12
    D3DMULTISAMPLE_13_SAMPLES = 13
    D3DMULTISAMPLE_14_SAMPLES = 14
    D3DMULTISAMPLE_15_SAMPLES = 15
    D3DMULTISAMPLE_16_SAMPLES = 16 (&H10)
End Enum
```

Constants

D3DMULTISAMPLE_NONE
No level of full-scene multisampling is available.

D3DMULTISAMPLE_2_SAMPLES through D3DMULTISAMPLE_16_SAMPLES
The level of full-scene multisampling available.

Remarks

In addition to enabling full-scene multisampling at **Direct3DDevice8.Reset** time, there will be render states that turn various aspects on and off at fine-grained levels.

Multisampling is valid only on a swap chain that is being created or reset with the D3DSWAPEFFECT_DISCARD swap effect.

See Also

Direct3D8.CheckDeviceMultiSampleType, **Direct3DDevice8.CreateDepthStencilSurface**, **Direct3DDevice8.CreateRenderTarget**, **D3DPRESENT_PARAMETERS**, **D3DSURFACE_DESC**

CONST_D3DORDERTYPE

#Defines the order type of a high-order surface.

```
Enum CONST_D3DORDERTYPE
    D3DORDER_LINEAR    = 1
    D3DORDER_CUBIC     = 3
```

IDH_CONST_D3DORDERTYPE_graphicsvb

```
    D3DORDER_QUINTIC   = 5
End Enum
```

Constants

D3DORDER_LINEAR
Linear order type.

D3DORDER_CUBIC
Cubic order type.

D3DORDER_QUINTIC
Quintic order type.

See Also

D3DRECTPATCH_INFO, **D3DTRIPATCH_INFO**

CONST_D3DPBLENDCAPSFLAGS

[#]Defines source-blending capabilities.

```
Enum CONST_D3DPBLENDCAPSFLAGS
    D3DPBLENDCAPS_ZERO          =    1
    D3DPBLENDCAPS_ONE           =    2
    D3DPBLENDCAPS_SRCCOLOR      =    4
    D3DPBLENDCAPS_INVSRCCOLOR   =    8
    D3DPBLENDCAPS_SRCALPHA      =   16 (&H10)
    D3DPBLENDCAPS_INVSRCALPHA   =   32 (&H20)
    D3DPBLENDCAPS_DESTALPHA     =   64 (&H40)
    D3DPBLENDCAPS_INVDESTALPHA  =  128 (&H80)
    D3DPBLENDCAPS_DESTCOLOR     =  256 (&H100)
    D3DPBLENDCAPS_INVDESTCOLOR  =  512 (&H200)
    D3DPBLENDCAPS_SRCALPHASAT   = 1024 (&H400)
    D3DPBLENDCAPS_BOTHSRCALPHA  = 2048 (&H800)
    D3DPBLENDCAPS_BOTHINVSRCALPHA = 4096 (&H1000)
End Enum
```

Constants

D3DPBLENDCAPS_ZERO
Blend factor is (0, 0, 0, 0).

D3DPBLENDCAPS_ONE
Blend factor is (1, 1, 1, 1).

D3DPBLENDCAPS_SRCCOLOR

[#] IDH_CONST_D3DPBLENDCAPSFLAGS_graphicsvb

Blend factor is (R_s, G_s, B_s, A_s).

D3DPBLENDCAPS_INVSRCCOLOR
Blend factor is (1–R_d, 1–G_d, 1–B_d, 1–A_d).

D3DPBLENDCAPS_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DPBLENDCAPS_INVSRCALPHA
Blend factor is (1–A_s, 1–A_s, 1–A_s, 1–A_s).

D3DPBLENDCAPS_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DPBLENDCAPS_INVDESTALPHA
Blend factor is (1–A_d, 1–A_d, 1–A_d, 1–A_d).

D3DPBLENDCAPS_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DPBLENDCAPS_INVDESTCOLOR
Blend factor is (1–R_d, 1–G_d, 1–B_d, 1–A_d).

D3DPBLENDCAPS_SRCALPHASAT
Blend factor is (f, f, f, 1); f = min(A_s, 1-A_d).

D3DPBLENDCAPS_BOTHSRCALPHA
The driver supports the D3DBLEND_BOTHSRCALPHA blend mode. (This blend mode is obsolete. For more information, see **CONST_D3DBLEND**.)

D3DPBLENDCAPS_BOTHINVSRCALPHA
Source blend factor is (1–A_s, 1–A_s, 1–A_s, 1–A_s), and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden.

Remarks

The RGBA values of the source and destination are indicated by the subscripts *s* and *d*.

The values in this enumeration define the values for the **SrcBlendCaps** member of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DPATCHEDGESTYLE

#Defines source-blending capabilities.

```
Enum CONST_D3DPATCHEDGESTYLE
    D3DPATCHEDGE_DISCRETE = 0,
    D3DPATCHEDGE_CONTINUOUS = 1
```

IDH_CONST_D3DPATCHEDGESTYLE_graphicsvb

```
End Enum
```

Constants

D3DPATCHEDGE_DISCRETE
: Discrete edge style. In discrete mode, you can specify float tessellation but it will be truncated to integers.

D3DPATCHEDGE_CONTINUOUS
: Continuous edge style. In continuous mode, tessellation is specified as float values which can be smoothly varied to reduce "popping" artifacts.

Remarks

Note that continuous tessellation produces a completely different tessellation pattern from the discrete one for the same tessellation values (this is more apparent in wire-frame mode). Thus, 4.0 continuous is not the same as 4 discrete.

CONST_D3DPCMPCAPSFLAGS

[#]Defines comparison capabilities.

```
Enum CONST_D3DPCMPCAPSFLAGS
   D3DPCMPCAPS_NEVER       =  1
   D3DPCMPCAPS_LESS        =  2
   D3DPCMPCAPS_EQUAL       =  4
   D3DPCMPCAPS_LESSEQUAL   =  8
   D3DPCMPCAPS_GREATER     = 16 (&H10)
   D3DPCMPCAPS_NOTEQUAL    = 32 (&H20)
   D3DPCMPCAPS_GREATEREQUAL = 64 (&H40)
   D3DPCMPCAPS_ALWAYS      = 128 (&H80)
End Enum
```

Constants

D3DPCMPCAPS_NEVER
: Always fail the test.

D3DPCMPCAPS_LESS
: Pass the test if the new value is less than the current value.

D3DPCMPCAPS_EQUAL
: Pass the test if the new value equals the current value.

D3DPCMPCAPS_LESSEQUAL
: Pass the test if the new value is less than or equal to the current value.

D3DPCMPCAPS_GREATER

IDH_CONST_D3DPCMPCAPSFLAGS_graphicsvb

Pass the test if the new value is greater than the current value.

D3DPCMPCAPS_NOTEQUAL
Pass the test if the new value does not equal the current value.

D3DPCMPCAPS_GREATEREQUAL
Pass the test if the new value is greater than or equal to the current value.

D3DPCMPCAPS_ALWAYS
Always pass the test.

Remarks

The values in this enumeration define the supported compare functions for the **AlphaCmpCaps** and **ZCmpCaps** members of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DPMISCCAPSFLAGS

#Defines general capabilities for a primitive.

```
Enum CONST_D3DPMISCCAPSFLAGS
   D3DPMISCCAPS_MASKZ              =    2
   D3DPMISCCAPS_LINEPATTERNREP     =    4
   D3DPMISCCAPS_CULLNONE           =   16 (&H10)
   D3DPMISCCAPS_CULLCW             =   32 (&H20)
   D3DPMISCCAPS_CULLCCW            =   64 (&H40)
   D3DPMISCCAPS_COLORWRITEENABLE   =  128 (&H80)
   D3DPMISCCAPS_CLIPPLANESCALEDPOINTS =  256 (&H100)
   D3DPMISCCAPS_CLIPTLVERTS        =  512 (&H200)
   D3DPMISCCAPS_TSSARGTEMP         = 1024 (&H400)
   D3DPMISCCAPS_BLENDOP            = 2048 (&H800)
End Enum
```

Constants

D3DPMISCCAPS_MASKZ
The device can enable and disable modification of the depth buffer on pixel operations.

D3DPMISCCAPS_LINEPATTERNREP
The driver can handle values other than 1 in the **RepeatFactor** member of the **D3DLINEPATTERN** type. (This applies only to line-drawing primitives.)

D3DPMISCCAPS_CULLNONE

IDH_CONST_D3DPMISCCAPSFLAGS_graphicsvb

The driver does not perform triangle culling. This corresponds to the D3DCULL_NONE member of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_CULLCW
The driver supports clockwise triangle culling through the D3DRS_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CW member of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_CULLCCW
The driver supports counterclockwise culling through the D3DRS_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CCW member of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_COLORWRITEENABLE
The device supports per-channel writes for the render target color buffer through the D3DRS_COLORWRITEENABLE state.

D3DPMISCCAPS_CLIPPLANESCALEDPOINTS
Device correctly clips scaled points of size greater than 1.0 to user-defined clipping planes.

D3DPMISCCAPS_CLIPTLVERTS
Device clips post-transformed vertex primitives.

D3DPMISCCAPS_TSSARGTEMP
Device supports D3DTA_TEMP for temporary register.

D3DPMISCCAPS_BLENDOP
Device supports the alpha blending operations defined in the CONST_D3DBLENDOP enumerated type.

Remarks

MiscCaps

The values in this enumeration define the values for the **PrimitveMiscCaps** member of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DPOOL

#Defines the memory class that holds a resource's buffers.

```
Enum CONST_D3DPOOL
   D3DPOOL_DEFAULT       = 0
   D3DPOOL_MANAGED       = 1
   D3DPOOL_SYSTEMMEM     = 2
End Enum
```

IDH_CONST_D3DPOOL_graphicsvb

Constants

D3DPOOL_DEFAULT

Resources are placed in the memory pool most appropriate for the set of usages requested for the given resource. This is usually video memory, including both local video memory and AGP memory. The D3DPOOL_DEFAULT pool is separate from D3DPOOL_MANAGED and D3DPOOL_SYTEMMEM and specifies that the resource will be placed in the preferred memory for device access. Note that D3DPOOL_DEFAULT never indicates that either D3DPOOL_MANAGED or D3DPOOL_SYSTEMMEM should be chosen as the memory pool type for this resource. Textures placed in the D3DPOOL_DEFAULT pool cannot be locked and are therefore not directly accessible. Instead you must use functions such as **Direct3DDevice8.CopyRects** and **Direct3dDevice8.UpdateTexture**. D3DPOOL_MANAGED is probably a better choice than D3DPOOL_DEFAULT for most applications. Note that some textures created in driver proprietary pixel formats, unknown to the Direct3D runtime, can be locked. Also note that—unlike textures—swapchain back-buffers, render targets, vertex buffers, and index buffers can be locked.

When a device becomes lost, resources created using D3DPOOL_DEFAULT must be released before calling **Direct3DDevice8.Reset**. See Lost Devices for more information.

D3DPOOL_MANAGED

Resources will be copied automatically to device-accessible memory as needed. Managed resources are backed by system memory, and do not need to be recreated when a device is lost. See Managing Resources for more information. Managed resources can be locked. Only the system-memory copy is directly modified. Direct3D copies your changes to driver accessible memory as needed.

D3DPOOL_SYSTEMMEM

Memory that is not typically accessible by the 3-D device. This flag consumes system RAM but does not reduce pageable RAM. These resources do not need to be recreated when a device is lost. Resources in this pool can be locked, and they can be used as the source for a **Direct3DDevice8.CopyRects** or **Direct3DDevice8.UpdateTexture** operation to a memory resource created with D3DPOOL_DEFAULT.

Remarks

Pools cannot be mixed for different objects contained within one resource (mip levels in a mipmap), and once a pool is chosen it cannot be changed.

Applications should use D3DPOOL_MANAGED for most static resources, as this isolates the application from dealing with lost device situations. (Managed resources are restored by the runtime.) This is especially beneficial for UMA systems. Dynamic texture resources can also be a good match for D3DPOOL_MANAGED, in spite of

the high frequency at which they change. Other dynamic resources are not a good match for D3DPOOL_MANAGED. In fact, index buffers and vertex buffers cannot be created using D3DPOOL_MANAGED together with D3DUSAGE_DYNAMIC.

For dynamic textures, it is sometimes desirable to use a pair of video-memory and system-memory textures, allocating the video memory using D3DPOOL_DEFAULT and the system memory using D3DPOOL_SYSTEMMEM. You can lock and modify the bits of the system memory texture using a locking method. Then you can update the video memory texture using **Direct3DDevice8.UpdateTexture**.

See Also

Direct3DDevice8.CreateCubeTexture, **Direct3DDevice8.CreateIndexBuffer**, **Direct3DDevice8.CreateTexture**, **Direct3DDevice8.CreateVertexBuffer**, **Direct3DDevice8.CreateVolumeTexture**, **D3DINDEXBUFFER_DESC**, **D3DSURFACE_DESC**, **D3DVERTEXBUFFER_DESC**, **D3DVOLUME_DESC**

CONST_D3DPRASTERCAPSFLAGS

#Defines constants that provide information on raster-drawing capabilities.

```
Enum CONST_D3DPRASTERCAPSFLAGS
   D3DPRASTERCAPS_DITHER                =      1
   D3DPRASTERCAPS_PAT                   =      8
   D3DPRASTERCAPS_ZTEST                 =     16 (&H10)
   D3DPRASTERCAPS_FOGVERTEX             =    128 (&H80)
   D3DPRASTERCAPS_FOGTABLE              =    256 (&H100)
   D3DPRASTERCAPS_ANTIALIASEDGES        =   4096 (&H1000)
   D3DPRASTERCAPS_MIPMAPLODBIAS         =   8192 (&H2000)
   D3DPRASTERCAPS_ZBIAS                 =  16384 (&H4000)
   D3DPRASTERCAPS_ZBUFFERLESSHSR        =  32768 (&H8000)
   D3DPRASTERCAPS_FOGRANGE              =  65536 (&H10000)
   D3DPRASTERCAPS_ANISOTROPY            = 131072 (&H20000)
   D3DPRASTERCAPS_WBUFFER               = 262144 (&H40000)
   D3DPRASTERCAPS_WFOG                  = 1048576 (&H100000)
   D3DPRASTERCAPS_ZFOG                  = 2097152 (&H200000)
   D3DPRASTERCAPS_COLORPERSPECTIVE      = 4194304
(&H400000)
   D3DPRASTERCAPS_STRETCHBLTMULTISAMPLE = 8388608
(&H800000)
End Enum
```

IDH_CONST_D3DPRASTERCAPSFLAGS_graphicsvb

Constants

D3DPRASTERCAPS_DITHER
The device can dither to improve color resolution.

D3DPRASTERCAPS_PAT
The driver can perform patterned drawing lines or fills with D3DRS_LINEPATTERN for the primitive being queried.

D3DPRASTERCAPS_ZTEST
The device can perform z-test operations. This effectively renders a primitive and indicates whether any z pixels have been rendered.

D3DPRASTERCAPS_FOGVERTEX
The device calculates the fog value during the lighting operation, places the value into the alpha component of the **specular** member of the **D3DTLVERTEX** type and interpolates the fog value during rasterization.

D3DPRASTERCAPS_FOGTABLE
The device calculates the fog value by referring to a lookup table containing fog values that are indexed to the depth of a given pixel.

D3DPRASTERCAPS_ANTIALIASEDGES
The device can antialias lines forming the convex outline of objects. For more information, see D3DRS_EDGEANTIALIAS in the **CONST_D3DRENDERSTATETYPE** enumeration.

D3DPRASTERCAPS_MIPMAPLODBIAS
The device supports level-of-detail (LOD) bias adjustments. These bias adjustments enable an application to make a mipmap appear crisper or less sharp than it normally would. For more information about LOD bias in mipmaps, see D3DTSS_MIPMAPLODBIAS.

D3DPRASTERCAPS_ZBIAS
The device supports z-bias values. These are integer values assigned to polygons that enable physically coplanar polygons to appear separate. For more information, see D3DRS_ZBIAS in the **CONST_D3DRENDERSTATETYPE** enumeration.

D3DPRASTERCAPS_ZBUFFERLESSHSR
The device can perform hidden-surface removal (HSR) without requiring the application to sort polygons and without requiring the allocation of a depth-buffer. This leaves more video memory for textures. The method used to perform HSR is hardware-dependent and is transparent to the application.

Z-bufferless HSR is performed if no depth-buffer surface is associated with the rendering-target surface and the depth-buffer comparison test is enabled (that is, when the state value associated with the D3DRS_ZENABLE enumeration constant is set to TRUE).

D3DPRASTERCAPS_FOGRANGE
The device supports range-based fog. In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene. For more information, see Range-Based Fog.

D3DPRASTERCAPS_ANISOTROPY
The device supports anisotropic filtering.

D3DPRASTERCAPS_WBUFFER
The device supports depth buffering using w. For more information, see Depth Buffers.

D3DPRASTERCAPS_WFOG
The device supports w-based fog. W-based fog is used when a perspective projection matrix is specified, but affine projections still use z-based fog. The system considers a projection matrix that contains a nonzero value in the [3][4] element to be a perspective projection matrix.

D3DPRASTERCAPS_ZFOG
The device supports z-based fog.

D3DPRASTERCAPS_COLORPERSPECTIVE
The device iterates colors perspective correct.

D3DPRASTERCAPS_STRETCHBLTMULTISAMPLE
The device provides limited multisample support through a stretch-blt implementation. When this capability is set, D3DRS_MULTISAMPLEANTIALIAS cannot be turned on and off in the middle of a scene. Multisample masking cannot be performed if this flag is set.

Remarks

These flags may be combined and present in the **RasterCaps** member of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DPRESENT_INTERVAL_FLAGS

#Maximum rate at which the swap chain's back buffers may be presented.

```
Enum CONST_D3DPRESENT_INTERVAL_FLAGS
   D3DPRESENT_INTERVAL_DEFAULT   =          0
   D3DPRESENT_INTERVAL_ONE       =          1
   D3DPRESENT_INTERVAL_TWO       =          2
   D3DPRESENT_INTERVAL_THREE     =          4
   D3DPRESENT_INTERVAL_FOUR      =          8
   D3DPRESENT_INTERVAL_IMMEDIATE =  -2147483648 (&H80000000)
End Enum
```

IDH_CONST_D3DPRESENT_INTERVAL_FLAGS_graphicsvb

Constants

D3DPRESENT_INTERVAL_DEFAULT
The driver supports the default presentation interval.

D3DPRESENT_INTERVAL_ONE
The driver will wait for the vertical retrace period. Present operations will not be affected more frequently than the screen refresh.

D3DPRESENT_INTERVAL_TWO
The driver will wait for the vertical retrace period. Present operations will not be affected more frequently than every second screen refresh.

D3DPRESENT_INTERVAL_THREE
The driver will wait for the vertical retrace period. Present operations will not be affected more frequently than every third screen refresh.

D3DPRESENT_INTERVAL_FOUR
The driver will wait for the vertical retrace period. Present operations will not be affected more frequently than every fourth screen refresh.

D3DPRESENT_INTERVAL_IMMEDIATE
Present operations may be affected immediately. The driver will not wait for the vertical retrace period.

Remarks

For a windowed swap chain, this value must be D3DPRESENT_INTERVAL_DEFAULT (0). For a full-screen swap chain it may be D3DPRESENT_INTERVAL_DEFAULT or the value corresponding to exactly one of the flags enumerated in the **PresentationIntervals** member of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DPRESENT_RATE_FLAGS

#Defines the maximal rate at which frames can be presented to the output.

```
Enum CONST_D3DPRESENT_RATE_FLAGS
   D3DPRESENT_RATE_DEFAULT   =         0
   D3DPRESENT_RATE_UNLIMITED = 2147483647 (&H7FFFFFFF)
End Enum
```

IDH_CONST_D3DPRESENT_RATE_FLAGS_graphicsvb

Constants

D3DPRESENT_RATE_UNLIMITED
The presentation rate runs as quickly as the hardware can deliver frames.

D3DPRESENT_RATE_DEFAULT
Microsoft® Direct3D® will choose the presentation rate, or adopt the current rate if windowed.

Remarks

One of these flags may be specified in the **FullScreen_RefreshRateInHz** member of **D3DPRESENT_PARAMETERS**.

See Also

D3DPRESENT_PARAMETERS

CONST_D3DPRIMITIVETYPE

#Defines the primitives supported by Microsoft® Direct3D®.

```
Enum CONST_D3DPRIMITIVETYPE
   D3DPT_POINTLIST     = 1
   D3DPT_LINELIST      = 2
   D3DPT_LINESTRIP     = 3
   D3DPT_TRIANGLELIST  = 4
   D3DPT_TRIANGLESTRIP = 5
   D3DPT_TRIANGLEFAN   = 6
End Enum
```

Constants

D3DPT_POINTLIST
Renders the vertices as a collection of isolated points.

D3DPT_LINELIST
Renders the vertices as a list of isolated straight line segments. Calls using this primitive type fail if the count is less than 2 or is odd.

D3DPT_LINESTRIP
Renders the vertices as a single polyline. Calls using this primitive type fail if the count is less than 2.

D3DPT_TRIANGLELIST
Renders the specified vertices as a sequence of isolated triangles. Each group of three vertices defines a separate triangle.

Backface culling is affected by the current winding-order render state.

IDH_CONST_D3DPRIMITIVETYPE_graphicsvb

D3DPT_TRIANGLESTRIP
Renders the vertices as a triangle strip. The backface-culling flag is automatically flipped on even-numbered triangles.

D3DPT_TRIANGLEFAN
Renders the vertices as a triangle fan.

Remarks

Using triangle strips or fans is often more efficient than using triangle lists because fewer vertices are duplicated. For more information see Triangle Strips and Triangle Fans.

See Also

Direct3DDevice8.DrawIndexedPrimitive, **Direct3DDevice8.DrawIndexedPrimitiveUP**, **Direct3DDevice8.DrawPrimitive**, **Direct3DDevice8.DrawPrimitiveUP**

CONST_D3DPSHADECAPSFLAGS

#Defines shading operations capabilities.

```
Enum CONST_D3DPSHADECAPSFLAGS
   D3DPSHADECAPS_COLORGOURAUDRGB     =      8
   D3DPSHADECAPS_SPECULARGOURAUDRGB  =    512 (&H200)
   D3DPSHADECAPS_ALPHAGOURAUDBLEND   =  16384 (&H4000)
   D3DPSHADECAPS_FOGGOURAUD          = 524288 (&H80000)
End Enum
```

Constants

D3DPSHADECAPS_COLORGOURAUDRGB
Device can support colored Gouraud shading in the RGB color model. In this mode, the color component for a primitive is provided at vertices and interpolated across a face along with the other color components. In the RGB lighting model, the red, green, and blue components are interpolated.

D3DPSHADECAPS_SPECULARGOURAUDRGB
Device can support specular highlights in Gouraud shading in the RGB color model.

D3DPSHADECAPS_ALPHAGOURAUDBLEND
Device can support an alpha component for Gouraud-blended transparency (the D3DSHADE_GOURAUD state for the **CONST_D3DSHADEMODE** enumeration). In this mode, the alpha color component of a primitive is provided at vertices and interpolated across a face along with the other color components.

IDH_CONST_D3DPSHADECAPSFLAGS_graphicsvb

D3DPSHADECAPS_FOGGOURAUD
Device can support fog in the Gouraud shading mode.

Remarks

These flags may be combined and present in the **ShadeCaps** member of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DPTADDRESSCAPSFLAGS

#Defines texture-addressing capabilities.

```
Enum CONST_D3DPTADDRESSCAPSFLAGS
   D3DPTADDRESSCAPS_WRAP         = 1
   D3DPTADDRESSCAPS_MIRROR       = 2
   D3DPTADDRESSCAPS_CLAMP        = 4
   D3DPTADDRESSCAPS_BORDER       = 8
   D3DPTADDRESSCAPS_INDEPENDENTUV = 16 (&H10)
   D3DPTADDRESSCAPS_MIRRORONCE    = 32 (&H20)
End Enum
```

Constants

D3DPTADDRESSCAPS_WRAP
Device can wrap textures to addresses.

D3DPTADDRESSCAPS_MIRROR
Device can mirror textures to addresses.

D3DPTADDRESSCAPS_CLAMP
Device can clamp textures to addresses.

D3DPTADDRESSCAPS_BORDER
Device supports setting coordinates outside the range [0.0, 1.0] to the border color, as specified by the D3DTSS_BORDERCOLOR texture-stage state.

D3DPTADDRESSCAPS_INDEPENDENTUV
Device can separate the texture-addressing modes of the u and v coordinates of the texture. This ability corresponds to the D3DTSS_ADDRESSU and D3DTSS_ADDRESSV render-state values.

D3DPTADDRESSCAPS_MIRRORONCE
Device can take the absolute value of the texture coordinate (thus, mirroring around 0), and then clamp to the maximum value.

IDH_CONST_D3DPTADDRESSCAPSFLAGS_graphicsvb

Remarks

These flags may be combined and present in the **TextureAddressCaps** member of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DPTEXTURECAPSFLAGS

#Defines miscellaneous texture-mapping capabilities.

```
Enum CONST_D3DPTEXTURECAPSFLAGS
   D3DPTEXTURECAPS_PERSPECTIVE          =     1
   D3DPTEXTURECAPS_POW2                 =     2
   D3DPTEXTURECAPS_ALPHA                =     4
   D3DPTEXTURECAPS_SQUAREONLY           =     32 (&H20)
   D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE =     64 (&H40)
   D3DPTEXTURECAPS_ALPHAPALETTE         =    128 (&H80)
   D3DPTEXTURECAPS_NONPOW2CONDITIONAL   =    256 (&H100)
   D3DPTEXTURECAPS_PROJECTED            =   1024 (&H400)
   D3DPTEXTURECAPS_CUBEMAP              =   2048 (&H800)
   D3DPTEXTURECAPS_VOLUMEMAP            =   8192 (&H2000)
   D3DPTEXTURECAPS_MIPMAP               =  16384 (&H4000)
   D3DPTEXTURECAPS_MIPVOLUMEMAP         =  32768 (&H8000)
   D3DPTEXTURECAPS_MIPCUBEMAP           =  65536 (&H10000)
   D3DPTEXTURECAPS_CUBEMAP_POW2         = 131072 (&H20000)
   D3DPTEXTURECAPS_VOLUMEMAP_POW2       = 262144 (&H40000)
End Enum
```

Constants

D3DPTEXTURECAPS_PERSPECTIVE
: Perspective correction texturing is supported.

D3DPTEXTURECAPS_POW2
: All textures must have widths and heights specified as powers of 2. This requirement does not apply to either cube textures or volume textures.

D3DPTEXTURECAPS_ALPHA
: Alpha in texture pixels is supported.

D3DPTEXTURECAPS_SQUAREONLY

IDH_CONST_D3DPTEXTURECAPSFLAGS_graphicsvb

All textures must be square.

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE
Texture indices are not scaled by the texture size prior to interpolation.

D3DPTEXTURECAPS_ALPHAPALETTE
Device can draw alpha from texture palettes.

D3DPTEXTURECAPS_NONPOW2CONDITIONAL
Conditionally supports the use of textures with dimensions that are not powers of 2. A device that exposes this capability can use such a texture if all of the following requirements are met.

- The texture addressing mode for the texture stage is set to D3DTADDRESS_CLAMP.
- Texture wrapping for the texture stage is disabled (D3DRS_WRAP*n* set to 0).
- Mipmapping is not in use (use magnification filter only).

D3DPTEXTURECAPS_PROJECTED
Supports the D3DTTFF_PROJECTED texture transformation flag. When applied, the device divides transformed texture coordinates by the last texture coordinate. If this capability is supported then the projective divide happens on a per pixel basis. If this capability is not set and the application still wants the projective divide to happen, then it happens on a per-vertex basis by the Direct3D runtime.

D3DPTEXTURECAPS_CUBEMAP
Supports cube textures.

D3DPTEXTURECAPS_VOLUMEMAP
Device supports volume textures.

D3DPTEXTURECAPS_MIPMAP
Device supports mipmapped textures.

D3DPTEXTURECAPS_MIPVOLUMEMAP
Device supports mipmapped volume textures.

D3DPTEXTURECAPS_MIPCUBEMAP
Device supports mipmapped cube textures.

D3DPTEXTURECAPS_CUBEMAP_POW2
Device requires that cube texture maps have dimensions specified as powers of 2.

D3DPTEXTURECAPS_VOLUMEMAP_POW2
Device requires that volume texture maps have dimensions specified as powers of 2.

Remarks

These flags may be combined and present in the **TextureCaps** member of **D3DCAPS8**.

See Also

D3DCAPS8

CONST_D3DPTFILTERCAPSFLAGS

#Defines texture-filtering capabilities.

```
Enum CONST_D3DPTFILTERCAPSFLAGS
    D3DPTFILTERCAPS_MINFPOINT        =       256 (&H100)
    D3DPTFILTERCAPS_MINFLINEAR       =       512 (&H200)
    D3DPTFILTERCAPS_MINFANISOTROPIC  =      1024 (&H400)
    D3DPTFILTERCAPS_MIPFPOINT        =     65536 (&H10000)
    D3DPTFILTERCAPS_MIPFLINEAR       =    131072 (&H20000)
    D3DPTFILTERCAPS_MAGFPOINT        =  16777216 (&H1000000)
    D3DPTFILTERCAPS_MAGFLINEAR       =  33554432 (&H2000000)
    D3DPTFILTERCAPS_MAGFANISOTROPIC  =  67108864 (&H4000000)
    D3DPTFILTERCAPS_MAGFAFLATCUBIC   = 134217728 (&H8000000)
    D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC = 268435456
(&H10000000)
End Enum
```

Constants

D3DPTFILTERCAPS_MINFPOINT
: The device supports per-stage point-sample filtering for minifying textures. The point-sample minification filter is represented by the D3DTEXF_POINT member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

D3DPTFILTERCAPS_MINFLINEAR
: The device supports per-stage bilinear interpolation filtering for minifying textures. The bilinear minification filter is represented by the D3DTEXF_LINEAR member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

D3DPTFILTERCAPS_MINFANISOTROPIC
: The device supports per-stage anisotropic filtering for minifying textures. The anisotropic minification filter is represented by the D3DTEXF_ANISOTROPIC member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

D3DPTFILTERCAPS_MIPFPOINT
: The device supports per-stage point-sample filtering for mipmaps. The point-sample mipmapping filter is represented by the D3DTEXF_POINT member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

D3DPTFILTERCAPS_MIPFLINEAR

IDH_CONST_D3DPTFILTERCAPSFLAGS_graphicsvb

The device supports per-stage trilinear interpolation filtering for mipmaps. The trilinear interpolation mipmapping filter is represented by the D3DTEXF_LINEAR member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

D3DPTFILTERCAPS_MAGFPOINT
The device supports per-stage point-sample filtering for magnifying textures. The point-sample magnification filter is represented by the D3DTEXF_POINT member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

D3DPTFILTERCAPS_MAGFLINEAR
The device supports per-stage bilinear interpolation filtering for magnifying textures. The bilinear interpolation magnification filter is represented by the D3DTEXF_LINEAR member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

D3DPTFILTERCAPS_MAGFANISOTROPIC
The device supports per-stage anisotropic filtering for magnifying textures. The anisotropic magnification filter is represented by the D3DTEXF_ANISOTROPIC member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

D3DPTFILTERCAPS_MAGFAFLATCUBIC
The device supports per-stage flat cubic filtering for magnifying textures. The flat cubic magnification filter is represented by the D3DTEXF_FLATCUBIC member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC
The device supports the per-stage Gaussian cubic filtering for magnifying textures. The Gaussian cubic magnification filter is represented by the D3DTEXF_GAUSSIANCUBIC member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration.

Remarks

These flags may be combined and present in the **TextureFilterCaps** member of **D3DCAPS8**.

The following flags concerning mipmapped textures are not supported in DirectX 8:

- D3DPTFILTERCAPS_NEAREST
- D3DPTFILTERCAPS_LINEAR
- D3DPTFILTERCAPS_MIPNEAREST
- D3DPTFILTERCAPS_MIPLINEAR
- D3DPTFILTERCAPS_LINEARMIPNEAREST
- D3DPTFILTERCAPS_LINEARMIPLINEAR

See Also

D3DCAPS8

CONST_D3DRENDERSTATETYPE

[#]Defines device render states

```
Enum CONST_D3DRENDERSTATETYPE
   D3DRS_ZENABLE              =  7
   D3DRS_FILLMODE             =  8
   D3DRS_SHADEMODE            =  9
   D3DRS_LINEPATTERN          =  10
   D3DRS_ZWRITEENABLE         =  14
   D3DRS_ALPHATESTENABLE      =  15
   D3DRS_LASTPIXEL            =  16 (&H10)
   D3DRS_SRCBLEND             =  19 (&H13)
   D3DRS_DESTBLEND            =  20 (&H14)
   D3DRS_CULLMODE             =  22 (&H16)
   D3DRS_ZFUNC                =  23 (&H17)
   D3DRS_ALPHAREF             =  24 (&H18)
   D3DRS_ALPHAFUNC            =  25 (&H19)
   D3DRS_DITHERENABLE         =  26 (&H1A)
   D3DRS_ALPHABLENDENABLE     =  27 (&H1B)
   D3DRS_FOGENABLE            =  28 (&H1C)
   D3DRS_SPECULARENABLE       =  29 (&H1D)
   D3DRS_ZVISIBLE             =  30 (&H1E)
   D3DRS_FOGCOLOR             =  34 (&H22)
   D3DRS_FOGTABLEMODE         =  35 (&H23)
   D3DRS_FOGSTART             =  36 (&H24)
   D3DRS_FOGEND               =  37 (&H25)
   D3DRS_FOGDENSITY           =  38 (&H26)
   D3DRS_EDGEANTIALIAS        =  40 (&H28)
   D3DRS_ZBIAS                =  47 (&H2F)
   D3DRS_RANGEFOGENABLE       =  48 (&H30)
   D3DRS_STENCILENABLE        =  52 (&H34)
   D3DRS_STENCILFAIL          =  53 (&H35)
   D3DRS_STENCILZFAIL         =  54 (&H36)
   D3DRS_STENCILPASS          =  55 (&H37)
   D3DRS_STENCILFUNC          =  56 (&H38)
   D3DRS_STENCILREF           =  57 (&H39)
   D3DRS_STENCILMASK          =  58 (&H3A)
   D3DRS_STENCILWRITEMASK     =  59 (&H3B)
   D3DRS_TEXTUREFACTOR        =  60 (&H3C)
   D3DRS_WRAP0                = 128 (&H80)
   D3DRS_WRAP1                = 129 (&H81)
   D3DRS_WRAP2                = 130 (&H82)
   D3DRS_WRAP3                = 131 (&H83)
```

[#] IDH_CONST_D3DRENDERSTATETYPE_graphicsvb

```
    D3DRS_WRAP4                 = 132 (&H84)
    D3DRS_WRAP5                 = 133 (&H85)
    D3DRS_WRAP6                 = 134 (&H86)
    D3DRS_WRAP7                 = 135 (&H87)
    D3DRS_CLIPPING               = 136 (&H88)
    D3DRS_LIGHTING               = 137 (&H89)
    D3DRS_AMBIENT                = 139 (&H8B)
    D3DRS_FOGVERTEXMODE          = 140 (&H8C)
    D3DRS_COLORVERTEX            = 141 (&H8D)
    D3DRS_LOCALVIEWER            = 142 (&H8E)
    D3DRS_NORMALIZENORMALS       = 143 (&H8F)
    D3DRS_DIFFUSEMATERIALSOURCE  = 145 (&H91)
    D3DRS_SPECULARMATERIALSOURCE = 146 (&H92)
    D3DRS_AMBIENTMATERIALSOURCE  = 147 (&H93)
    D3DRS_EMISSIVEMATERIALSOURCE = 148 (&H94)
    D3DRS_VERTEXBLEND            = 151 (&H97)
    D3DRS_CLIPPLANEENABLE        = 152 (&H98)
    D3DRS_SOFTWAREVERTEXPROCESSING = 153 (&H99)
    D3DRS_POINTSIZE              = 154 (&H9A)
    D3DRS_POINTSIZE_MIN          = 155 (&H9B)
    D3DRS_POINTSPRITEENABLE      = 156 (&H9C)
    D3DRS_POINTSCALEENABLE       = 157 (&H9D)
    D3DRS_POINTSCALE_A           = 158 (&H9E)
    D3DRS_POINTSCALE_B           = 159 (&H9F)
    D3DRS_POINTSCALE_C           = 160 (&HA0)
    D3DRS_MULTISAMPLEANTIALIAS   = 161 (&HA1)
    D3DRS_MULTISAMPLEMASK        = 162 (&HA2)
    D3DRS_PATCHEDGESTYLE         = 163 (&HA3)
    D3DRS_PATCHSEGMENTS          = 164 (&HA4)
    D3DRS_DEBUGMONITORTOKEN      = 165 (&HA5)
    D3DRS_POINTSIZE_MAX          = 166 (&HA6)
    D3DRS_INDEXVERTEXBLENDENABLE = 167 (&HA7)
    D3DRS_COLORWRITEENABLE       = 168 (&HA8)
    D3DRS_TWEENFACTOR            = 170 (&HAA)
    D3DRS_BLENDOP                = 171 (&HAB)
End Enum
```

Constants

D3DRS_ZENABLE

Depth-buffering state as one member of the **CONST_D3DZBUFFERTYPE** enumeration. Set this state to D3DZB_TRUE to enable z-buffering, D3DZB_USEW to enable w-buffering, or D3DZB_FALSE to disable depth buffering.

The default value for this render state is D3DZB_TRUE if a depth stencil was created along with the swap chain by setting the **EnableAutoDepthStencil** member of the **D3DPRESENT_PARAMETERS** structure to TRUE, and D3DZB_FALSE otherwise.

D3DRS_FILLMODE
One or more members of the **CONST_D3DFILLMODE** enumeration. The default value is D3DFILL_SOLID.

D3DRS_SHADEMODE
One or more members of the **CONST_D3DSHADEMODE** enumeration. The default value is D3DSHADE_GOURAUD.

D3DRS_LINEPATTERN
D3DLINEPATTERN type. The default values are 0 for **wRepeatPattern**, and 0 for **wLinePattern**.

D3DRS_ZWRITEENABLE
TRUE to enable the application to write to the depth buffer. The default value is TRUE. This member enables an application to prevent the system from updating the depth buffer with new depth values. If FALSE, depth comparisons are still made according to the render state D3DRS_ZFUNC, assuming that depth buffering is taking place, but depth values are not written to the buffer.

D3DRS_ALPHATESTENABLE
TRUE to enable alpha tests. The default value is FALSE. This member enables applications to turn off the tests that accept or reject a pixel, based on its alpha value.

The incoming alpha value is compared with the reference alpha value, using the comparison function provided by the D3DRS_ALPHAFUNC render state. When this mode is enabled, alpha blending occurs only if the test succeeds.

D3DRS_LASTPIXEL
FALSE to enable drawing the last pixel in a line or triangle. The default value is TRUE.

D3DRS_SRCBLEND
One member of the **CONST_D3DBLEND** enumeration. The default value is D3DBLEND_ONE.

D3DRS_DESTBLEND
One member of the **CONST_D3DBLEND** enumeration. The default value is D3DBLEND_ZERO.

D3DRS_CULLMODE
Specifies how back-facing triangles are culled, if at all. This can be set to one member of the **CONST_D3DCULL** enumerated type. The default value is D3DCULL_CCW.

D3DRS_ZFUNC
One member of the **CONST_D3DCMPFUNC** enumeration. The default value is D3DCMP_LESSEQUAL. This member enables an application to accept or reject a pixel, based on its distance from the camera.

The depth value of the pixel is compared with the depth-buffer value. If the depth value of the pixel passes the comparison function, the pixel is written.

The depth value is written to the depth buffer only if the render state is TRUE.

Software rasterizers and many hardware accelerators work faster if the depth test fails, because there is no need to filter and modulate the texture if the pixel is not going to be rendered.

D3DRS_ALPHAREF
Value specifying a reference alpha value against which pixels are tested when alpha-testing is enabled. This can be a 16:16 fixed point value (D3DFIXED) ranging from 0 to 1, inclusive, where 1.0 is represented as &H00010000. The default value is 0.0.

D3DRS_ALPHAFUNC
One member of the **CONST_D3DCMPFUNC** enumeration. The default value is D3DCMP_ALWAYS. This member enables an application to accept or reject a pixel, based on its alpha value.

D3DRS_DITHERENABLE
TRUE to enable dithering. The default value is FALSE.

D3DRS_ALPHABLENDENABLE
TRUE to enable alpha-blended transparency. The default value is FALSE.

The type of alpha blending is determined by the D3DRS_SRCBLEND and D3DRS_DESTBLEND render states.

Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **DevCaps** member of the **D3DCAPS8** type to find out whether this render state is supported.

D3DRS_FOGENABLE
TRUE to enable fog blending. The default value is FALSE.

D3DRS_SPECULARENABLE
TRUE to enable specular highlights. The default value is FALSE.

Specular highlights are calculated as though every vertex in the object being lit were at the object's origin. This gives the expected results as long as the object is modeled around the origin and the distance from the light to the object is relatively large.

When this member is set to TRUE, the specular color is added to the base color after the texture cascade but before alpha blending.

D3DRS_ZVISIBLE
Not supported.

D3DRS_FOGCOLOR
Value of type **Long** indicating RGB color. The default value is 0.

D3DRS_FOGTABLEMODE
The fog formula to be used for pixel fog. Set to one of the members of the **CONST_D3DFOGMODE** enumeration. The default value is D3DFOG_NONE.

D3DRS_FOGSTART

Depth at which pixel or vertex fog effects begin for linear fog mode. Depth is specified in world space for vertex fog, and either device space [0.0, 1.0] or world space for pixel fog. For pixel fog, these values are in device space when the system uses z for fog calculations, and world-space when the system is using eye-relative fog (w-fog).

D3DRS_FOGEND

Depth at which pixel or vertex fog effects end for linear fog mode. Depth is specified in world space for vertex fog, and either device space [0.0, 1.0] or world space for pixel fog. For pixel fog, these values are in device space when the system uses z for fog calculations, and world-space when the system is using eye-relative fog (w-fog).

D3DRS_FOGDENSITY

Fog density for pixel or vertex fog used in the exponential fog modes (D3DFOG_EXP and D3DFOG_EXP2). Valid density values range from 0.0 through 1.0. The default value is 1.0.

D3DRS_EDGEANTIALIAS

TRUE to antialias lines forming the convex outline of objects. The default value is FALSE. If TRUE, applications should render only lines, and only to the exterior edges of polygons in a scene. The behavior is undefined if triangles or points are drawn when this render state is set. Antialiasing is performed by averaging the values of neighboring pixels. Although this is not the best way to perform antialiasing, it can be very efficient; hardware that supports this kind of operation is becoming more common.

You can enable edge antialiasing only on devices that expose the D3DPRASTERCAPS_ANTIALIASEDGES capability.

D3DRS_ZBIAS

An integer value in the range 0 to 16 that causes polygons that are physically coplanar to appear separate. Polygons with a high z-bias value will appear in front of polygons with a low value, without requiring sorting for drawing order. Polygons with a value of 1 appear in front of polygons with a value of 0, and so on. The default value is zero.

D3DRS_RANGEFOGENABLE

TRUE to enable range-based vertex fog. The default value is FALSE, in which case the system uses depth-based fog. In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene. In range-based fog, all fog methods work as usual, except that they use range instead of depth in the computations.

Range is the correct factor to use for fog computations, but depth is commonly used instead because range is expensive to compute and depth is generally already available. Using depth to calculate fog has the undesirable effect of having the fogginess of peripheral objects change as the viewer's eye moves—in this case, the depth changes, and the range remains constant.

Because no hardware currently supports per-pixel range-based fog, range correction is offered only for vertex fog.

D3DRS_STENCILENABLE

TRUE to enable stenciling, or FALSE to disable stenciling. The default value is FALSE.

D3DRS_STENCILFAIL
Stencil operation to perform if the stencil test fails. This can be one member of the **CONST_D3DSTENCILOP** enumeration. The default value is D3DSTENCILOP_KEEP.

D3DRS_STENCILZFAIL
Stencil operation to perform if the stencil test passes and the depth test (z-test) fails. This can be one of the members of the **CONST_D3DSTENCILOP** enumeration. The default value is D3DSTENCILOP_KEEP.

D3DRS_STENCILPASS
Stencil operation to perform if both the stencil and the depth (z) tests pass. This can be one member of the **CONST_D3DSTENCILOP** enumeration. The default value is D3DSTENCILOP_KEEP.

D3DRS_STENCILFUNC
Comparison function for the stencil test. This can be one member of the **CONST_D3DCMPFUNC** enumeration. The default value is D3DCMP_ALWAYS.

The comparison function is used to compare the reference value to a stencil buffer entry. This comparison applies only to the bits in the reference value and stencil buffer entry that are set in the stencil mask (set by the D3DRS_STENCILMASK render state). If TRUE, the stencil test passes.

D3DRS_STENCILREF
Integer reference value for the stencil test. The default value is 0.

D3DRS_STENCILMASK
Mask applied to the reference value and each stencil buffer entry to determine the significant bits for the stencil test. The default mask is &HFFFFFFFF.

D3DRS_STENCILWRITEMASK
Mask applied to the reference value and each stencil buffer entry to determine the significant bits for the stencil test. The default mask is &HFFFFFFFF.

D3DRS_TEXTUREFACTOR
Color used for multiple-texture blending with the D3DTA_TFACTOR texture-blending argument or the D3DTOP_BLENDFACTORALPHA texture-blending operation.

D3DRS_WRAP0 through D3DRS_WRAP7
Texture-wrapping behavior for multiple sets of texture coordinates. Valid values for these render states can be any combination of the D3DWRAPCOORD_0 (or D3DWRAP_U), D3DWRAPCOORD_1 (or D3DWRAP_V), D3DWRAPCOORD_2 (or D3DWRAP_W), and D3DWRAPCOORD_3 flags defined by the **CONST_D3DWRAPFLAGS** enumeration. These cause the system to wrap in the direction of the first, second, third, and fourth dimensions, sometimes referred to at the s, t, r, and q directions, for a given texture. The default value for these render states is 0 (wrapping disabled in all directions).

D3DRS_CLIPPING

TRUE to enable primitive clipping by Microsoft® Direct3D®, or FALSE to disable it. The default value is TRUE.

D3DRS_LIGHTING
TRUE to enable Direct3D lighting, or FALSE to disable it. The default value is TRUE. Only vertices that include a vertex normal are properly lit; vertices that do not contain a normal employ a dot product of 0 in all lighting calculations.

D3DRS_AMBIENT
Ambient light color. The default value is 0.

D3DRS_FOGVERTEXMODE
Fog formula to be used for vertex fog. Set to one member of the **CONST_D3DFOGMODE** enumeration. The default value is D3DFOG_NONE. This render state is analogous to the legacy D3DLIGHTSTATE_FOGVERTEXMODE lighting state.

D3DRS_COLORVERTEX
TRUE to enable per-vertex color, or FALSE to disable it. The default value is TRUE. Enabling per-vertex color allows the system to include the color defined for individual vertices in its lighting calculations.

For more information, see the following render states.

- D3DRS_DIFFUSEMATERIALSOURCE
- D3DRS_SPECULARMATERIALSOURCE
- D3DRS_AMBIENTMATERIALSOURCE
- D3DRS_EMISSIVEMATERIALSOURCE

D3DRS_LOCALVIEWER
TRUE to enable camera-relative specular highlights, or FALSE to use orthogonal specular highlights. The default value is TRUE. Applications that use orthogonal projection should specify FALSE.

D3DRS_NORMALIZENORMALS
TRUE to enable automatic normalization of vertex normals, or FALSE to disable it. The default value is FALSE. Enabling this feature causes the system to normalize the vertex normals for vertices after transforming them to camera space, which can be computationally expensive.

D3DRS_DIFFUSEMATERIALSOURCE
Diffuse color source for lighting calculations. Valid values are members of the **CONST_D3DMATERIALCOLORSOURCE** enumeration. The default value is D3DMCS_COLOR1. The value for this render state is used only if the D3DRS_COLORVERTEX render state is set to TRUE.

D3DRS_SPECULARMATERIALSOURCE
Specular color source for lighting calculations. Valid values are members of the **CONST_D3DMATERIALCOLORSOURCE** enumeration. The default value is D3DMCS_COLOR2.

D3DRS_AMBIENTMATERIALSOURCE

Ambient color source for lighting calculations. Valid values are members of the **CONST_D3DMATERIALCOLORSOURCE** enumeration. The default value is D3DMCS_COLOR2.

D3DRS_EMISSIVEMATERIALSOURCE
Emissive color source for lighting calculations. Valid values are members of the **CONST_D3DMATERIALCOLORSOURCE** enumeration. The default value is D3DMCS_MATERIAL.

D3DRS_VERTEXBLEND
Number of matrices to use to perform geometry blending, if any. Valid values are members of the **CONST_D3DVERTEXBLENDFLAGS** enumeration. The default value is D3DVBF_DISABLE.

D3DRS_CLIPPLANEENABLE
Enables or disables user-defined clipping planes. Valid values are combinations of values from the **CONST_D3DCLIPPLANEFLAGS** enumeration. If you include a value from the enumeration, the corresponding clipping plane is enabled; if a value is not included, the clipping plane is disabled. The default value is D3DCPF_DISABLEALL.

D3DRS_SOFTWAREVERTEXPROCESSING
Value that enables applications to query and select hardware or software vertex processing. For a D3DDEVTYPE_SW device type this value is fixed to TRUE. This value can be set by the application for a D3DDEVTYPE_REF device type. For a D3DDEVTYPE_HAL device type, this value can be set only by the application when D3DDEVCAPS_HWTRANSFORMANDLIGHT is set; otherwise this flag is fixed to TRUE. When variable, the default value is FALSE.

Changing the vertex processing render state in mixed vertex processing mode will reset the current stream, indices, and vertex shader to their default values of ByVal 0 or 0

D3DRS_POINTSIZE
Value that specifies the size to use for point size computation in cases where point size is not specified for each vertex. This value is not used when the vertex contains point size. This value is in screen space units if D3DRS_POINTSCALEENABLE is FALSE; otherwise this value is in world space units. The default value is 1.0. The range for this value is greater than or equal to 0.0.

D3DRS_POINTSIZE_MIN
Value that specifies the minimum size of point primitives. Point primitives are clamped to this size during rendering. Setting this to values smaller than 1.0 results in points dropping out when the point does not cover a pixel center and antialiasing is disabled or being rendered with reduced intensity when antialiasing is enabled. The default value is 1.0. The range for this value is greater than or equal to 0.0.

D3DRS_POINTSPRITEENABLE
When TRUE, texture coordinates of point primitives are set so that full textures are mapped on each point. When FALSE, the vertex texture coordinates are used for the entire point. The default value is FALSE. You can achieve DirectX 7 style

single-pixel points by setting D3DRS_POINTSCALEENABLE to FALSE and D3DRS_POINTSIZE to 1.0, which are the default values.

D3DRS_POINTSCALEENABLE
Value that controls computation of size for point primitives. When TRUE, the point size is interpreted as a camera space value and is scaled by the distance function and the frustum to viewport Y axis scaling to compute the final screen space point size. When FALSE, the point size is interpreted as screen space and used directly. The default value is FALSE.

D3DRS_POINTSCALE_A
Value that controls for distance-based size attenuation for point primitives. Active only when D3DRS_POINTSCALEENABLE is TRUE. The default value is 1.0. The range for this value is greater than or equal to 0.0.

D3DRS_POINTSCALE_B
Value that controls for distance-based size attenuation for point primitives. Active only when D3DRS_POINTSCALEENABLE is TRUE. The default value is 0.0. The range for this value is greater than or equal to 0.0.

D3DRS_POINTSCALE_C
Value that controls for distance-based size attenuation for point primitives. Active only when D3DRS_POINTSCALEENABLE is TRUE. The default value is 0.0. The range for this value is greater than or equal to 0.0.

D3DRS_MULTISAMPLEANTIALIAS
Value that determines how individual samples are computed when using a multisample render target buffer. When set to TRUE, the multiple samples are computed so that full-scene antialiasing is performed by sampling at different sample positions for each multiple sample. When set to FALSE, the multiple samples are all written with the same sample value (sampled at the pixel center), which enables non-antialiased rendering to a multisample buffer. This render state has no effect when rendering to a single sample buffer. The default value is TRUE.

D3DRS_MULTISAMPLEMASK
Each bit in this mask, starting at the LSB, controls modification of one of the samples in a multisample render target. Thus, for an 8 sample render target, the low byte contains the 8 write enables for each of the 8 samples. This render state has no effect when rendering to a single sample buffer. The default value is &HFFFFFFFF.

This render state enables use of a multisample buffer as an accumulation buffer, doing multipass rendering of geometry where each pass updates a subset of samples.

D3DRS_PATCHEDGESTYLE
Sets whether patch edges will use float-style tessellation. Possible values are defined by the CONST_D3DPATCHEDGESTYLE enumerated type.

D3DRS_PATCHSEGMENTS
Number of segments per edge when drawing high-order primitives.

D3DRS_DEBUGMONITORTOKEN
Set only for debugging the monitor.

D3DRS_POINTSIZE_MAX
Value that specifies the maximum size to which point sprites will be clamped. The value must be less than or equal to the **MaxPointSize** member of **D3DCAPS8** and greater than or equal to D3DRS_POINTSIZE_MIN.

D3DRS_INDEXVERTEXBLENDENABLE
When TRUE, enables indexed vertex blending. When FALSE, disables indexed vertex blending. If this render state is enabled, the user must pass matrix indices as a packed **LONG** with every vertex. When the render state is disabled and vertex blending is enabled through the D3DRS_VERTEXBLEND state, it is equivalent to having matrix indices 0, 1, 2, 3 in every vertex.

D3DRS_COLORWRITEENABLE
Value that enables a per-channel write for the render target color buffer. A set bit results in the color channel being updated during 3-D rendering. A clear bit results in the color channel being unaffected. This functionality is available if the D3DPMISCCAPS_COLORWRITEENABLE capabilities bit is set in the **PrimitiveMiscCaps** member of the **D3DCAPS8** type for the device. This render state does not affect the clear operation. The default value is &H0000000F.

Valid values for this render state can be any combination of values defined by the **CONST_D3DCOLORWRITEENABLEFLAGS** enumeration.

D3DRS_TWEENFACTOR
Value that controls the tween factor.

D3DRS_BLENDOP
Value used to select the arithmetic operation applied when the alpha blending render state, D3DRS_ALPHABLENDENABLE, is set to TRUE. Valid values are defined by the **CONST_D3DBLENDOP** enumerated type. The default value is D3DBLENDOP_ADD.

If the D3DPMISCCAPS_BLENDOP device capability is not supported, then D3DBLENDOP_ADD is performed.

Remarks

Direct3D defines the D3DRENDERSTATE_WRAPBIAS constant defined by the **CONST_D3DWRAPBIAS** enumeration as a convenience for applications to enable or disable texture wrapping based on the zero-based integer of a texture coordinate set (rather than explicitly using one of the D3DRS_WRAP*n* state values). Add the D3DRENDERSTATE_WRAPBIAS value to the zero-based index of a texture coordinate set to calculate the D3DRS_WRAP*n* value that corresponds to that index, as shown in the following example:

```
On Local Error Resume Next

' Enable U/V wrapping for textures that use the texture
' coordinate set at the index within the lIndex variable.
Call d3dDevice8.SetRenderState( _
    lIndex + D3DRENDERSTATE_WRAPBIAS, _
    D3DWRAPCOORD_0 Or D3DWRAPCOORD_1)
```

```
' If lIndex is 3, the value that results from
' the addition equates to D3DRENDERSTATE_WRAP3 (131).
If Err.Number <> DD_OK Then
  ' Code to handle error goes here.
End If
```

CONST_D3DRESOURCETYPE

#Defines resource types.

```
Enum CONST_D3DRESOURCETYPE
   D3DRTYPE_SURFACE       = 1
   D3DRTYPE_VOLUME        = 2
   D3DRTYPE_TEXTURE       = 3
   D3DRTYPE_VOLUMETEXTURE = 4
   D3DRTYPE_CUBETEXTURE   = 5
   D3DRTYPE_VERTEXBUFFER  = 6
   D3DRTYPE_INDEXBUFFER   = 7
End Enum
```

Constants

D3DRTYPE_SURFACE
: Surface resource.

D3DRTYPE_VOLUME
: Volume resource.

D3DRTYPE_TEXTURE
: Texture resource.

D3DRTYPE_VOLUMETEXTURE
: Volume texture resource.

D3DRTYPE_CUBETEXTURE
: Cube texture resource.

D3DRTYPE_VERTEXBUFFER
: Vertex buffer resource.

D3DRTYPE_INDEXBUFFER
: Index buffer resource.

See Also

Direct3DResource8.GetType

IDH_CONST_D3DRESOURCETYPE_graphicsvb

CONST_D3DSCPFLAGS

#Defines update options for the cursor.

```
Enum CONST_D3DSCPFLAGS
    D3DCURSOR_IMMEDIATE_UPDATE = 1
End Enum
```

Constants

D3DCURSOR_IMMEDIATE_UPDATE
Update cursor at the refresh rate.

See Also

Direct3DDevice8.SetCursorPosition

CONST_D3DSGRFLAGS

#Defines constants that indicate whether or not gamma correction should be applied.

```
Enum CONST_D3DSGRFLAGS
   D3DSGR_NO_CALIBRATION = 0
   D3DSGR_CALIBRATE      = 1
End Enum
```

Constants

D3DSGR_NO_CALIBRATION
No gamma correction is applied. The supplied gamma table is transferred directly to the device.

D3DSGR_CALIBRATE
If a gamma calibrator is installed, the ramp will be modified before being sent to the device to account for the system and monitor response curves.

If a calibrator is not installed, the ramp will be passed directly to the device.

See Also

Direct3DDevice8.SetGammaRamp

IDH_CONST_D3DSCPFLAGS_graphicsvb
IDH_CONST_D3DSGRFLAGS_graphicsvb

CONST_D3DSHADEMODE

#Defines constants that describe the supported shade mode.

```
Enum CONST_D3DSHADEMODE
   D3DSHADE_FLAT    = 1
   D3DSHADE_GOURAUD = 2
   D3DSHADE_PHONG   = 3
End Enum
```

Constants

D3DSHADE_FLAT
: Flat shade mode. The color and specular component of the first vertex in the triangle are used to determine the color and specular component of the face. These colors remain constant across the triangle; that is, they are not interpolated. The specular alpha is interpolated. See Remarks.

D3DSHADE_GOURAUD
: Gouraud shade mode. The color and specular components of the face are determined by a linear interpolation between all three of the triangle's vertices.

D3DSHADE_PHONG
: Phong shade mode is not currently supported.

Remarks

The first vertex of a triangle for flat shading mode is defined in the following manner.

- For a triangle list, the first vertex of the triangle *i* is $i * 3$.
- For a triangle strip, the first vertex of the triangle *i* is vertex *i*.
- For a triangle fan, the first vertex of the triangle *i* is vertex $i + 1$.

These flags are used to set the value of the D3DRS_SHADEMODE render state for the **CONST_D3DRENDERSTATETYPE** enumeration.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DSPDFLAGS

#Defines how private data should be managed.

```
Enum CONST_D3DSPDFLAGS
```

IDH_CONST_D3DSHADEMODE_graphicsvb
IDH_CONST_D3DSPDFLAGS_graphicsvb

```
    D3DSPD_IUNKNOWN = 1
End Enum
```

Constants

D3DSPD_IUNKNOWN
When the passed data is no longer needed, Microsoft® Direct3D® automatically releases the memory associated with the data.

See Also

Direct3DResource8.SetPrivateData, **Direct3DSurface8.SetPrivateData**, **Direct3DVolume8.SetPrivateData**

CONST_D3DSTATEBLOCKTYPE

#Defines logical groups of device states.

```
Enum CONST_D3DSTATEBLOCKTYPE
    D3DSBT_ALL        = 1
    D3DSBT_PIXELSTATE  = 2
    D3DSBT_VERTEXSTATE = 3
End Enum
```

Constants

D3DSBT_ALL
Capture all device states.

All current render states.	All current clipplanes.
All current texture stage states.	The current material.
All current textures.	All current lights and enabled light parameters.
The current palette.	The current pixel shader.
All current streams.	The current pixel shader constants.
The current viewport.	The current vertex shader.
All current transforms.	The current vertex shader constants.

D3DSBT_PIXELSTATE
Capture all of the following pixel-related device states.

Render States

D3DRS_ALPHABLENDENABLE	D3DRS_ALPHAFUNC
D3DRS_ALPHAREF	D3DRS_ALPHATESTENABLE

IDH_CONST_D3DSTATEBLOCKTYPE_graphicsvb

D3DRS_BLENDOP
D3DRS_COLORWRITEENABLE
D3DRS_DESTBLEND
D3DRS_DITHERENABLE
D3DRS_EDGEANTIALIAS
D3DRS_FILLMODE
D3DRS_FOGDENSITY
D3DRS_FOGEND
D3DRS_FOGSTART
D3DRS_LASTPIXEL
D3DRS_LINEPATTERN
D3DRS_SHADEMODE
D3DRS_SRCBLEND
D3DRS_STENCILENABLE
D3DRS_STENCILFAIL
D3DRS_STENCILFUNC
D3DRS_STENCILMASK
D3DRS_STENCILPASS
D3DRS_STENCILREF
D3DRS_STENCILWRITEMASK
D3DRS_STENCILZFAIL
D3DRS_TEXTUREFACTOR
D3DRS_WRAP0 through D3DRS_WRAP7
D3DRS_ZBIAS
D3DRS_ZENABLE
D3DRS_ZFUNC
D3DRS_ZWRITEENABLE

Texture Stage States

D3DTSS_ADDRESSU
D3DTSS_ADDRESSV
D3DTSS_ADDRESSW
D3DTSS_ALPHAARG0
D3DTSS_ALPHAARG1
D3DTSS_ALPHAARG2
D3DTSS_ALPHAOP
D3DTSS_BORDERCOLOR
D3DTSS_BUMPENVLOFFSET
D3DTSS_BUMPENVLSCALE
D3DTSS_BUMPENVMAT00
D3DTSS_BUMPENVMAT01
D3DTSS_BUMPENVMAT10
D3DTSS_BUMPENVMAT11
D3DTSS_COLORARG0
D3DTSS_COLORARG1
D3DTSS_COLORARG2
D3DTSS_COLOROP
D3DTSS_MAGFILTER
D3DTSS_MAXANISOTROPY
D3DTSS_MAXMIPLEVEL
D3DTSS_MINFILTER
D3DTSS_MIPFILTER
D3DTSS_MIPMAPLODBIAS
D3DTSS_RESULTARG
D3DTSS_TEXCOORDINDEX
D3DTSS_TEXTURETRANSFORMFLAGS

D3DSBT_VERTEXSTATE
Capture all the current lights, the current vertex shader and vertex shader constants, and the texture stage states specified by D3DTSS_TEXCOORDINDEX and D3DTSS_TEXTURETRANSFORMFLAGS. In addition, D3DSBT_VERTEXSTATE captures all of the following vertex-related device states.

Render States

D3DRS_AMBIENT	D3DRS_AMBIENTMATERIALSOURCE
D3DRS_CLIPPING	D3DRS_CLIPPLANEENABLE
D3DRS_COLORVERTEX	D3DRS_DIFFUSEMATERIALSOURCE
D3DRS_EMISSIVEMATERIALSOURCE	D3DRS_FOGDENSITY
D3DRS_FOGEND	D3DRS_FOGSTART
D3DRS_FOGTABLEMODE	D3DRS_FOGVERTEXMODE
D3DRS_INDEXVERTEXBLENDENABLE	D3DRS_LIGHTING
D3DRS_LOCALVIEWER	D3DRS_MULTISAMPLEANTIALIAS
D3DRS_MULTISAMPLEMASK	D3DRS_NORMALIZENORMALS
D3DRS_PATCHEDGESTYLE	D3DRS_PATCHSEGMENTS
D3DRS_POINTSCALE_A	D3DRS_POINTSCALE_B
D3DRS_POINTSCALE_C	D3DRS_POINTSCALEENABLE
D3DRS_POINTSIZE	D3DRS_POINTSIZE_MAX
D3DRS_POINTSIZE_MIN	D3DRS_POINTSPRITEENABLE
D3DRS_RANGEFOGENABLE	D3DRS_SOFTWAREVERTEXPROCESSING
D3DRS_SPECULARMATERIALSOURCE	D3DRS_TWEENFACTOR
D3DRS_VERTEXBLEND	

Remarks

The D3DSBT_PIXELSTATE and D3DSBT_VERTEXSTATE values identify different logical groups of device states, though some states are common to both groups. The union of D3DSBT_PIXELSTATE and D3DSBT_VERTEXSTATE is not equal to D3DSBT_ALL. The D3DSBT_PIXELSTATE and D3DSBT_VERTEXSTATE values enable the capture of these frequently modified states between calls to **Direct3Ddevice8.DrawPrimitive** without incurring the performance penalty of capturing the entire state.

See Also

Direct3DDevice8.CreateStateBlock

CONST_D3DSTENCILCAPFLAGS

#Defines stencil buffer capability flags

```
Enum CONST_D3DSTENCILCAPFLAGS
   D3DSTENCILCAPS_KEEP    =  1
   D3DSTENCILCAPS_ZERO    =  2
   D3DSTENCILCAPS_REPLACE =  4
   D3DSTENCILCAPS_INCRSAT =  8
   D3DSTENCILCAPS_DECRSAT = 16 (&H10)
   D3DSTENCILCAPS_INVERT  = 32 (&H20)
   D3DSTENCILCAPS_INCR    = 64 (&H40)
   D3DSTENCILCAPS_DECR    = 128 (&H80)
End Enum
```

Constants

D3DSTENCILCAPS_KEEP
The **D3DSTENCILOP_KEEP** operation is supported.

D3DSTENCILCAPS_ZERO
The **D3DSTENCILOP_ZERO** operation is supported.

D3DSTENCILCAPS_REPLACE
The **D3DSTENCILOP_REPLACE** operation is supported.

D3DSTENCILCAPS_INCRSAT
The **D3DSTENCILOP_INCRSAT** operation is supported.

D3DSTENCILCAPS_DECRSAT
The **D3DSTENCILOP_DECRSAT** operation is supported.

D3DSTENCILCAPS_INVERT
The **D3DSTENCILOP_INVERT** operation is supported.

D3DSTENCILCAPS_INCR
The **D3DSTENCILOP_INCR** operation is supported.

D3DSTENCILCAPS_DECR
The **D3DSTENCILOP_DECR** operation is supported.

Remarks

These flags may be combined and present in the **StencilCaps** member of **D3DCAPS8**.

For a definition of stencil operations, see **CONST_D3DSTENCILOP**.

IDH_CONST_D3DSTENCILCAPFLAGS_graphicsvb

See Also

D3DCAPS8, **CONST_D3DSTENCILOP**

CONST_D3DSTENCILOP

#Defines constants that define stencil operations.

```
Enum CONST_D3DSTENCILOP
    D3DSTENCILOP_KEEP    = 1
    D3DSTENCILOP_ZERO    = 2
    D3DSTENCILOP_REPLACE = 3
    D3DSTENCILOP_INCRSAT = 4
    D3DSTENCILOP_DECRSAT = 5
    D3DSTENCILOP_INVERT  = 6
    D3DSTENCILOP_INCR    = 7
    D3DSTENCILOP_DECR    = 8
End Enum
```

Constants

D3DSTENCILOP_KEEP
: Do not update the entry in the stencil buffer. This is the default value.

D3DSTENCILOP_ZERO
: Set the stencil-buffer entry to zero.

D3DSTENCILOP_REPLACE
: Replace the stencil-buffer entry with reference value.

D3DSTENCILOP_INCRSAT
: Increment the stencil-buffer entry, clamping to the maximum value. See Remarks for information on the maximum stencil-buffer values.

D3DSTENCILOP_DECRSAT
: Decrement the stencil-buffer entry, clamping to zero.

D3DSTENCILOP_INVERT
: Invert the bits in the stencil-buffer entry.

D3DSTENCILOP_INCR
: Increment the stencil-buffer entry, wrapping to zero if the new value exceeds the maximum value. See Remarks for information on the maximum stencil-buffer values.

D3DSTENCILOP_DECR
: Decrement the stencil-buffer entry, wrapping to the maximum value if the new value is less than zero.

IDH_CONST_D3DSTENCILOP_graphicsvb

Remarks

Stencil-buffer entries are integer values ranging inclusively from 0 to 2^n -1, where *n* is the bit depth of the stencil buffer.

These flags are used to set the value of the D3DRS_STENCILFAIL, D3DRS_STENCILZFAIL, and D3DRS_STENCILPASS render states for the **CONST_D3DRENDERSTATETYPE** enumeration.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DSWAPEFFECT

#Defines swap effects.

```
Enum CONST_D3DSWAPEFFECT
   D3DSWAPEFFECT_DISCARD    = 1
   D3DSWAPEFFECT_FLIP       = 2
   D3DSWAPEFFECT_COPY       = 3
   D3DSWAPEFFECT_COPY_VSYNC = 4
End Enum
```

Constants

D3DSWAPEFFECT_DISCARD

When a swap chain is created with a swap effect of D3DSWAPEFFECT_FLIP, D3DSWAPEFFECT_COPY or D3DSWAPEFFECT_COPY_VSYNC, the runtime will guarantee that a **Direct3DDevice8.Present** operation will not affect the content of any of the back buffers. Unfortunately, meeting this guarantee can involve substantial video memory or processing overheads, especially when implementing flip semantics for a windowed swap chain or copy semantics for a full-screen swap chain. An application may use the D3DSWAPEFFECT_DISCARD swap effect to avoid these overheads and to enable the display driver to select the most efficient presentation technique for the swap chain. This is also the only swap effect that may be used when specifying a value other than D3DMULTISAMPLE_NONE for the **MultiSampleType** member of **D3DPRESENT_PARAMETERS**.

As with a swap chain that uses D3DSWAPEFFECT_FLIP, a swap chain that uses D3DSWAPEFFECT_DISCARD might include more than one back buffer, any of which may be accessed using **Direct3DDevice8.GetBackBuffer** or **Direct3DSwapChain8.GetBackBuffer**. The swap chain is best envisaged as a queue in which 0 always indexes the back buffer that will be displayed by the next Present operation and from which buffers are discarded once they have been displayed.

IDH_CONST_D3DSWAPEFFECT_graphicsvb

An application that uses this swap effect cannot make any assumptions about the contents of a discarded back buffer and should therefore update an entire back buffer before invoking a **Present** operation that would display it. Although this is not enforced, the debug version of the runtime will overwrite the contents of discarded back buffers with random data to enable developers to verify that their applications are updating the entire back buffer surfaces correctly.

For a full-screen swap chain, the presentation rate is determined by the value assigned to the **FullScreen_PresentationInterval** member of the **D3DPRESENT_PARAMETERS** structure when the device or swap chain is created. Unless this value is D3DPRESENT_INTERVAL_IMMEDIATE, the presentation will be synchronized with the vertical sync of the monitor. For a windowed swap chain, the presentation is implemented by means of copy operations and always occurs immediately.

D3DSWAPEFFECT_FLIP

The swap chain may comprise multiple back buffers and is best envisaged as a circular queue that includes the front buffer. Within this queue, the back buffers are always numbered sequentially from 0 to (N -1), where N is the number of back buffers, so that 0 denotes the least recently presented buffer. When **Present** is invoked, the queue is "rotated" so that the front buffer becomes back buffer (N - 1), while back buffer 0 becomes the new front buffer.

For a full-screen swap chain, the presentation rate is determined by the value assigned to the **FullScreen_PresentationInterval** field of the D3DPRESENT_PARAMETERS structure when the device or swap chain is created. Unless this value is D3DPRESENT_INTERVAL_IMMEDIATE, the presentation will be synchronized with the vertical sync of the monitor. For a windowed swap chain, the flipping is implemented by means of copy operations and the presentation always occurs immediately.

D3DSWAPEFFECT_COPY

This swap effect may be specified only for a swap chain comprising a single back buffer. Whether the swap chain is windowed or full-screen, the runtime will guarantee the semantics implied by a copy-based **Present** operation, namely that the operation leaves the content of the back buffer unchanged, instead of replacing it with the content of the front buffer as a flip-based **Present** operation would.

For a windowed swap chain, a **Present** operation causes the back buffer content to be copied to the client area of the target window immediately. No attempt is made to synchronize the copy with the vertical retrace period of the display adapter, so "tearing" effects may be observed. Windowed applications may wish to use D3DSWAPEFFECT_COPY_VSYNC instead to eliminate, or at least minimize, such tearing effects.

For a full-screen swap chain, the runtime uses a combination of flip operations and copy operations, supported if necessary by hidden back buffers, to accomplish the **Present** operation. Accordingly, the presentation is synchronized with the display adapter's vertical retrace and its rate is constrained by the chosen presentation interval. A swap chain specified with the D3DPRESENT_INTERVAL_IMMEDIATE flag is the only exception. (Refer to

the description of the **FullScreen_PresentationInterval** member of the **D3DPRESENT_PARAMETERS** structure.) In this case, a **Present** operation copies the back buffer content directly to the front buffer without waiting for the vertical retrace.

D3DSWAPEFFECT_COPY_VSYNC

Like D3DSWAPEFFECT_COPY, this swap effect may only be used with a swap chain comprising a single back buffer and guarantees that a **Present** operation applied to the swap chain will exhibit copy semantics, as described above for D3DSWAPEFFECT_COPY.

For a windowed swap chain, a **Present** operation causes the back buffer content to be copied to the client area of the target window. The runtime will attempt to eliminate tearing effects by avoiding the copy operation while the adapter is scanning within the destination rectangle on the display. It will also perform at most one such copy operation during the adapter's refresh period and thus limit the presentation frequency. Note, however, that if the adapter does not support the ability to report the raster status, the swap chain will behave as though it had been created with the D3DSWAPEFFECT_COPY swap effect. (Refer to the description of the D3DCAPS_READ_SCANLINE flag value for the **Caps** member of **D3DCAPS8**.)

For a full-screen swap chain, D3DSWAPEFFECT_COPY_VSYNC is identical to D3DSWAPEFFECT_COPY, except that the D3DPRESENT_INTERVAL_IMMEDIATE flag is meaningless when used in conjunction with D3DSWAPEFFECT_COPY_VSYNC. (Refer to the description of the **FullScreen_PresentationInterval** member of the **D3DPRESENT_PARAMETERS** structure.)

Remarks

The state of the back buffer after a call to **Present** is well-defined by each of these swap effects, and whether the Microsoft® Direct3D® device was created with a full-screen swap chain or a windowed swap chain has no effect on this state. In particular, the D3DSWAPEFFECT_FLIP swap effect operates the same whether windowed or full-screen, and the Direct3D run time guarantees this by creating extra buffers. As a result, it is recommended that applications use D3DSWAPEFFECT_DISCARD whenever possible to avoid any such penalties. This is because this swap effect will always be the most efficient in terms of memory consumption and performance.

See Also

Direct3DDevice8.Reset, **D3DPRESENT_PARAMETERS**

CONST_D3DTAFLAGS

#Defines texture argument flags used to define texture blending stages.

IDH_CONST_D3DTAFLAGS_graphicsvb

For details, see Texture Argument Flags.

CONST_D3DTEXOPCAPSFLAGS

[#]Defines texture-blending operation capabilities

```
Enum CONST_D3DTEXOPCAPSFLAGS
   D3DTEXOPCAPS_DISABLE                =       1
   D3DTEXOPCAPS_SELECTARG1             =       2
   D3DTEXOPCAPS_SELECTARG2             =       4
   D3DTEXOPCAPS_MODULATE               =       8
   D3DTEXOPCAPS_MODULATE2X             =      16 (&H10)
   D3DTEXOPCAPS_MODULATE4X             =      32 (&H20)
   D3DTEXOPCAPS_ADD                    =      64 (&H40)
   D3DTEXOPCAPS_ADDSIGNED              =     128 (&H80)
   D3DTEXOPCAPS_ADDSIGNED2X            =     256 (&H100)
   D3DTEXOPCAPS_SUBTRACT               =     512 (&H200)
   D3DTEXOPCAPS_ADDSMOOTH              =    1024 (&H400)
   D3DTEXOPCAPS_BLENDDIFFUSEALPHA      =    2048 (&H800)
   D3DTEXOPCAPS_BLENDTEXTUREALPHA      =    4096 (&H1000)
   D3DTEXOPCAPS_BLENDFACTORALPHA       =    8192 (&H2000)
   D3DTEXOPCAPS_BLENDTEXTUREALPHAPM    =   16384 (&H4000)
   D3DTEXOPCAPS_BLENDCURRENTALPHA      =   32768 (&H8000)
   D3DTEXOPCAPS_PREMODULATE            =   65536 (&H10000)
   D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR   =  131072
(&H20000)
   D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA   =  262144
(&H40000)
   D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR =  524288
(&H80000)
   D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA =  1048576
(&H100000)
   D3DTEXOPCAPS_BUMPENVMAP             =  2097152 (&H200000)
   D3DTEXOPCAPS_BUMPENVMAPLUMINANCE    =  4194304
(&H400000)
   D3DTEXOPCAPS_DOTPRODUCT3            =  8388608 (&H800000)
   D3DTEXOPCAPS_MULTIPLYADD            = 16777216 (&H1000000)
   D3DTEXOPCAPS_LERP                   = 33554432 (&H2000000)
End Enum
```

Constants

D3DTEXOPCAPS_DISABLE
The **D3DTOP_DISABLE** texture-blending operation is supported.

[#] IDH_CONST_D3DTEXOPCAPSFLAGS_graphicsvb

D3DTEXOPCAPS_SELECTARG1
The **D3DTOP_SELECTARG1** texture-blending operation is supported.

D3DTEXOPCAPS_SELECTARG2
The **D3DTOP_SELECTARG2** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATE
The **D3DTOP_MODULATE** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATE2X
The **D3DTOP_MODULATE2X** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATE4X
The **D3DTOP_MODULATE4X** texture-blending operation is supported.

D3DTEXOPCAPS_ADD
The **D3DTOP_ADD** texture-blending operation is supported.

D3DTEXOPCAPS_ADDSIGNED
The **D3DTOP_ADDSIGNED** texture-blending operation is supported.

D3DTEXOPCAPS_ADDSIGNED2X
The **D3DTOP_ADDSIGNED2X** texture-blending operation is supported.

D3DTEXOPCAPS_SUBTRACT
The **D3DTOP_SUBTRACT** texture-blending operation is supported.

D3DTEXOPCAPS_ADDSMOOTH
The **D3DTOP_ADDSMOOTH** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDDIFFUSEALPHA
The **D3DTOP_BLENDDIFFUSEALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDTEXTUREALPHA
The **D3DTOP_BLENDTEXTUREALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDFACTORALPHA
The **D3DTOP_BLENDFACTORALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDTEXTUREALPHAPM
The **D3DTOP_BLENDTEXTUREALPHAPM** texture-blending operation is supported.

D3DTEXOPCAPS_BLENDCURRENTALPHA
The **D3DTOP_BLENDCURRENTALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_PREMODULATE
The **D3DTOP_PREMODULATE** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR
The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA
The **D3DTOP_MODULATECOLOR_ADDALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR
The **D3DTOP_MODULATEINVALPHA_ADDCOLOR** texture-blending operation is supported.

D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA
The **D3DTOP_MODULATEINVCOLOR_ADDALPHA** texture-blending operation is supported.

D3DTEXOPCAPS_BUMPENVMAP
The **D3DTOP_BUMPENVMAP** texture-blending operation is supported.

D3DTEXOPCAPS_BUMPENVMAPLUMINANCE
The **D3DTOP_BUMPENVMAPLUMINANCE** texture-blending operation is supported.

D3DTEXOPCAPS_DOTPRODUCT3
The **D3DTOP_DOTPRODUCT3** texture-blending operation is supported.

D3DTEXOPCAPS_MULTIPLYADD
The **D3DTOP_MULTIPLYADD** texture-blending operation is supported.

D3DTEXOPCAPS_LERP
The D3DTOP_LERP texture-blending operation is supported.

Remarks

These flags may be combined and present in the **TextureOpCaps** member of **D3DCAPS8**.

For a definition of texture operations, see **CONST_D3DTEXTUREOP**.

See Also

D3DCAPS8, **CONST_D3DTEXTUREOP**

CONST_D3DTEXTUREADDRESS

#Defines constants that describe the supported texture-addressing modes.

```
Enum CONST_D3DTEXTUREADDRESS
   D3DTADDRESS_WRAP       = 1
   D3DTADDRESS_MIRROR     = 2
   D3DTADDRESS_CLAMP      = 3
   D3DTADDRESS_BORDER     = 4
   D3DTADDRESS_MIRRORONCE = 5
End Enum
```

Constants

D3DTADDRESS_WRAP

IDH_CONST_D3DTEXTUREADDRESS_graphicsvb

Tile the texture at every integer junction. For example, for u values between 0 and 3, the texture is repeated three times; no mirroring is performed.

D3DTADDRESS_MIRROR
Similar to D3DTADDRESS_WRAP, except that the texture is flipped at every integer junction. For u values between 0 and 1, for example, the texture is addressed normally; between 1 and 2, the texture is flipped (mirrored); between 2 and 3, the texture is normal again, and so on.

D3DTADDRESS_CLAMP
Texture coordinates outside the range [0.0, 1.0] are set to the texture color at 0.0 or 1.0, respectively.

D3DTADDRESS_BORDER
Texture coordinates outside the range [0.0, 1.0] are set to the border color.

D3DTADDRESS_MIRRORONCE
Similar to D3DTADDRESS_MIRROR and D3DTADDRESS_CLAMP. Takes the absolute value of the texture coordinate (thus mirroring around 0), and then clamps to the maximum value. The most common usage is for volume textures, where support for the full D3DTADDRESS_MIRRORONCE texture-addressing mode is not necessary, but the data is symmetric around the one axis.

Remarks

These flags are used to set the value of the D3DTSS_ADDRESSU, D3DTSS_ADDRESSV, and D3DTSS_ADDRESSW texture stage states for the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

See Also

CONST_D3DTEXTURESTAGESTATETYPE

CONST_D3DTEXTUREFILTERTYPE

#Defines texture filtering modes for a texture stage.

```
Enum CONST_D3DTEXTUREFILTERTYPE
   D3DTEXF_NONE         = 0
   D3DTEXF_POINT        = 1
   D3DTEXF_LINEAR       = 2
   D3DTEXF_ANISOTROPIC  = 3
   D3DTEXF_FLATCUBIC    = 4
   D3DTEXF_GAUSSIANCUBIC = 5
End Enum
```

IDH_CONST_D3DTEXTUREFILTERTYPE_graphicsvb

Constants

D3DTEXF_NONE
Mipmapping disabled. The rasterizer should use the magnification filter instead.

D3DTEXF_POINT
Point filtering used as a texture magnification or minification filter. The texel with coordinates nearest to the desired pixel value is used.

The texture filter to be used between mipmap levels is nearest-point mipmap filtering. The rasterizer uses the color from the texel of the nearest mipmap texture.

D3DTEXF_LINEAR
Bilinear interpolation filtering used as a texture magnification or minification filter. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

The texture filter to use between mipmap levels is trilinear mipmap interpolation. The rasterizer linearly interpolates pixel color, using the texels of the two nearest mipmap textures.

D3DTEXF_ANISOTROPIC
Anisotropic texture filtering used as a texture magnification or minification filter. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

D3DTEXF_FLATCUBIC
Flat-cubic filtering used as a texture magnification filter.

D3DTEXF_GAUSSIANCUBIC
Gaussian-cubic filtering used as a texture magnification filter.

Remarks

Not all valid filtering modes for a device will apply to volume maps. In general, D3DTEXF_POINT and D3DTEXF_LINEAR magnification filters will be supported for volume maps. If D3DPTEXTURECAPS_MIPVOLUMEMAP is set, then the D3DTEXF_POINT mipmap filter and D3DTEXF_POINT and D3DTEXF_LINEAR minification filters will be supported for volume maps. The device may or may not support the D3DTEXF_LINEAR mipmap filter for volume maps. DirectX 8.0 devices that support anisotropic filtering for 2-D maps do not necessarily support anisotropic filtering for volume maps. However, applications that enable anisotropic filtering will receive the best available filtering (probably linear) if anisotropic filtering is not supported.

Set a texture stage's magnification filter by calling the **Direct3DDevice8.SetTextureStageState** method with the D3DTSS_MAGFILTER render state value as the second parameter and one member of this enumeration as the third parameter.

Set a texture stage's minification filter by calling **SetTextureStageState** with the D3DTSS_MINFILTER render state value as the second parameter and one member

of this enumeration as the third parameter.

Set the texture filter to use between mipmap levels by calling **SetTextureStageState** with the D3DTSS_MIPFILTER render state value as the second parameter and one member of this enumeration as the third parameter.

See Also

D3DCAPS8, **CONST_D3DTEXTURESTAGESTATETYPE**

CONST_D3DTEXTUREOP

#Defines per-stage texture-blending operations.

```
Enum CONST_D3DTEXTUREOP
    D3DTOP_DISABLE              =  1
    D3DTOP_SELECTARG1            =  2
    D3DTOP_SELECTARG2            =  3
    D3DTOP_MODULATE             =  4
    D3DTOP_MODULATE2X            =  5
    D3DTOP_MODULATE4X            =  6
    D3DTOP_ADD                  =  7
    D3DTOP_ADDSIGNED             =  8
    D3DTOP_ADDSIGNED2X           =  9
    D3DTOP_SUBTRACT             = 10
    D3DTOP_ADDSMOOTH             = 11
    D3DTOP_BLENDDIFFUSEALPHA       = 12
    D3DTOP_BLENDTEXTUREALPHA       = 13
    D3DTOP_BLENDFACTORALPHA        = 14
    D3DTOP_BLENDTEXTUREALPHAPM     = 15
    D3DTOP_BLENDCURRENTALPHA       = 16 (&H10)
    D3DTOP_PREMODULATE           = 17 (&H11)
    D3DTOP_MODULATEALPHA_ADDCOLOR    = 18 (&H12)
    D3DTOP_MODULATECOLOR_ADDALPHA    = 19 (&H13)
    D3DTOP_MODULATEINVALPHA_ADDCOLOR = 20 (&H14)
    D3DTOP_MODULATEINVCOLOR_ADDALPHA = 21 (&H15)
    D3DTOP_BUMPENVMAP            = 22 (&H16)
    D3DTOP_BUMPENVMAPLUMINANCE     = 23 (&H17)
    D3DTOP_DOTPRODUCT3           = 24 (&H18)
    D3DTOP_MULTIPLYADD           = 25 (&H19)
    D3DTOP_LERP                 = 26 (&H1A)
End Enum
```

IDH_CONST_D3DTEXTUREOP_graphicsvb

Constants

Control members

D3DTOP_DISABLE

Disables output from this texture stage and all stages with a higher index. To disable texture mapping, set this as the color operation for the first texture stage (stage 0). Alpha operations cannot be disabled when color operations are enabled. Setting the alpha operation to D3DTOP_DISABLE when color blending is enabled causes undefined behavior.

D3DTOP_SELECTARG1

Use this texture stage's first color or alpha argument, unmodified, as the output. This operation affects the color argument when used with the D3DTSS_COLOROP texture-stage state, and the alpha argument when used with D3DTSS_ALPHAOP.

$S_{RGBA} = Arg1$

D3DTOP_SELECTARG2

Use this texture stage's second color or alpha argument, unmodified, as the output. This operation affects the color argument when used with the D3DTSS_COLOROP texture stage state, and the alpha argument when used with D3DTSS_ALPHAOP.

$S_{RGBA} = Arg2$

Modulation members

D3DTOP_MODULATE

Multiply the components of the arguments.

$S_{RGBA} = Arg1 \times Arg2$

D3DTOP_MODULATE2X

Multiply the components of the arguments, and shift the products to the left 1 bit (effectively multiplying them by 2) for brightening.

$S_{RGBA} = (Arg1 \times Arg2) << 1$

D3DTOP_MODULATE4X

Multiply the components of the arguments, and shift the products to the left 2 bits (effectively multiplying them by 4) for brightening.

$S_{RGBA} = (Arg1 \times Arg2) << 2$

Addition and subtraction members

D3DTOP_ADD

Add the components of the arguments.

$S_{RGBA} = Arg1 + Arg2$

D3DTOP_ADDSIGNED

Add the components of the arguments with a –0.5 bias, making the effective range of values from –0.5 through 0.5.

$S_{RGBA} = Arg1 + Arg2 - 0.5$

D3DTOP_ADDSIGNED2X

Add the components of the arguments with a –0.5 bias, and shift the products to the left 1 bit.

$S_{RGBA} = (Arg1 + Arg2 - 0.5) << 1$

D3DTOP_SUBTRACT

Subtract the components of the second argument from those of the first argument.

$S_{RGBA} = Arg1 - Arg2$

D3DTOP_ADDSMOOTH

Add the first and second arguments; then subtract their product from the sum.

$S_{RGBA} = Arg\ 1 + Arg2 - Arg1 \times Arg2$
$= Arg1 + Arg2\ (1 - Arg1\ )$

Linear alpha blending members

D3DTOP_BLENDDIFFUSEALPHA, D3DTOP_BLENDTEXTUREALPHA, D3DTOP_BLENDFACTORALPHA, and D3DTOP_BLENDCURRENTALPHA

Linearly blend this texture stage, using the interpolated alpha from each vertex (D3DTOP_BLENDDIFFUSEALPHA), alpha from this stage's texture (D3DTOP_BLENDTEXTUREALPHA), a scalar alpha (D3DTOP_BLENDFACTORALPHA) set with the D3DRS_TEXTUREFACTOR render state, or the alpha taken from the previous texture stage (D3DTOP_BLENDCURRENTALPHA).

$S_{RGBA} = Arg\ 1 \times (Alpha) + Arg2 \times (1 - Alpha\ )$

D3DTOP_BLENDTEXTUREALPHAPM
Linearly blend a texture stage that uses a premultiplied alpha.

$S_{RGBA} = Arg\ 1 + Arg2 \times (1 - Alpha)$

Specular mapping members

D3DTOP_PREMODULATE
Modulate this texture stage with the next texture stage.

D3DTOP_MODULATEALPHA_ADDCOLOR
Modulate the color of the second argument, using the alpha of the first argument; then add the result to argument one. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = Arg\ 1_{RGB} + Arg1_{A} \times Arg2_{RGB}$

D3DTOP_MODULATECOLOR_ADDALPHA
Modulate the arguments; then add the alpha of the first argument. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = Arg\ 1_{RGB} \times Arg2_{RGB} + Arg1_{A}$

D3DTOP_MODULATEINVALPHA_ADDCOLOR
Similar to D3DTOP_MODULATEALPHA_ADDCOLOR, but use the inverse of the alpha of the first argument. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = (1 - Arg\ 1_{A}) \times Arg2_{RGB} + Arg1_{RGB}$

D3DTOP_MODULATEINVCOLOR_ADDALPHA
Similar to D3DTOP_MODULATECOLOR_ADDALPHA, but use the inverse of the color of the first argument. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = (1 - Arg\ 1_{RGB}) \times Arg2_{RGB} + Arg1_{A}$

Bump-mapping members

D3DTOP_BUMPENVMAP
Perform per-pixel bump mapping, using the environment map in the next texture stage, without luminance. This operation is supported only for color operations

(D3DTSS_COLOROP).

D3DTOP_BUMPENVMAPLUMINANCE
Perform per-pixel bump mapping, using the environment map in the next texture stage, with luminance. This operation is supported only for color operations (D3DTSS_COLOROP).

D3DTOP_DOTPRODUCT3
Modulate the components of each argument as signed components, add their products; then replicate the sum to all color channels, including alpha. This operation is supported for color and alpha operations.

$$S_{RGBA} = (Arg\ 1_R \times Arg2_R + Arg1_G \times Arg2_G + Arg1_B \times Arg2_B)$$

In DirectX 6.0 and 7.0 multitexture operations the above inputs are all shifted down by half (y = x - 0.5) before use to simulate signed data, and the scalar result is automatically clamped to positive values and replicated to all three output channels. Also, note that as a color operation this does not updated the alpha it just updates the RGB components.

However, in DirectX 8.0 shaders, you can specify that the output be routed to the .rgb or the .a components or both (the default). You can also specify a separate scalar operation on the alpha channel.

Triadic texture blending members

D3DTOP_MULTIPLYADD
Performs a multiply-accumulate operation. It takes the last two arguments, multiplies them, adds them to the remaining input/source argument, and then places this sum into the result.

$S_{RGBA} = Arg1 + Arg2 * Arg3$

D3DTOP_LERP
Linearly interpolates between the 2nd and 3rd source arguments by a proportion specified in the 1st source argument.

$S_{RGBA} = (Arg1) * Arg2 + (1-Arg1) * Arg3$

Remarks

The members of this type are used when setting color or alpha operations by using the D3DTSS_COLOROP or D3DTSS_ALPHAOP texture stage state values with the **Direct3DDevice8.SetTextureStageState** method.

In the above formulas, S_{RGBA} is the RGBA color produced by a texture operation, and *Arg1* and *Arg2* represent the complete RGBA color of the texture arguments. Individual components of an argument are shown with subscripts. For example, the alpha component for argument 1 would be shown as $Arg1_A$.

See Also

CONST_D3DTEXTURESTAGESTATETYPE

CONST_D3DTEXTURESTAGESTATETYPE

#Defines texture stage states.

```
Enum CONST_D3DTEXTURESTAGESTATETYPE
   D3DTSS_COLOROP            =  1
   D3DTSS_COLORARG1          =  2
   D3DTSS_COLORARG2          =  3
   D3DTSS_ALPHAOP            =  4
   D3DTSS_ALPHAARG1          =  5
   D3DTSS_ALPHAARG2          =  6
   D3DTSS_BUMPENVMAT00       =  7
   D3DTSS_BUMPENVMAT01       =  8
   D3DTSS_BUMPENVMAT10       =  9
   D3DTSS_BUMPENVMAT11       = 10
   D3DTSS_TEXCOORDINDEX      = 11
   D3DTSS_ADDRESSU           = 13
   D3DTSS_ADDRESSV           = 14
   D3DTSS_BORDERCOLOR        = 15
   D3DTSS_MAGFILTER          = 16 (&H10)
   D3DTSS_MINFILTER          = 17 (&H11)
   D3DTSS_MIPFILTER          = 18 (&H12)
   D3DTSS_MIPMAPLODBIAS      = 19 (&H13)
   D3DTSS_MAXMIPLEVEL        = 20 (&H14)
   D3DTSS_MAXANISOTROPY      = 21 (&H15)
   D3DTSS_BUMPENVLSCALE      = 22 (&H16)
   D3DTSS_BUMPENVLOFFSET     = 23 (&H17)
   D3DTSS_TEXTURETRANSFORMFLAGS = 24 (&H18)
   D3DTSS_ADDRESSW           = 25 (&H19)
   D3DTSS_COLORARG0          = 26 (&H1A)
   D3DTSS_ALPHAARG0          = 27 (&H1B)
   D3DTSS_RESULTARG          = 28 (&H1C)
End Enum
```

Constants

D3DTSS_COLOROP

The texture-stage state is a texture color blending operation identified by one member of the **CONST_D3DTEXTUREOP** enumeration. The default value for

IDH_CONST_D3DTEXTURESTAGESTATETYPE_graphicsvb

the first texture stage (stage 0) is D3DTOP_MODULATE, and for all other stages the default is D3DTOP_DISABLE.

D3DTSS_COLORARG1
The texture-stage state is the first color argument for the stage, identified by one of the texture argument flags. The default argument is D3DTA_TEXTURE.

D3DTSS_COLORARG2
The texture-stage state is the second color argument for the stage, identified by one of the texture argument flags. The default argument is D3DTA_CURRENT.

D3DTSS_ALPHAOP
The texture-stage state is a texture alpha blending operation identified by one member of the **CONST_D3DTEXTUREOP** enumeration. The default value for the first texture stage (stage 0) is D3DTOP_SELECTARG1, and for all other stages the default is D3DTOP_DISABLE.

D3DTSS_ALPHAARG1
The texture-stage state is the first alpha argument for the stage, identified by one of the texture argument flags. The default argument is D3DTA_TEXTURE. If no texture is set for this stage, the default argument is D3DTA_DIFFUSE.

D3DTSS_ALPHAARG2
The texture-stage state is the second alpha argument for the stage, identified by one of the texture argument flags. The default argument is D3DTA_CURRENT.

D3DTSS_BUMPENVMAT00
The texture-stage state is a value for the [0][0] coefficient in a bump-mapping matrix. The default value is 0.0.

D3DTSS_BUMPENVMAT01
The texture-stage state is a value for the [0][1] coefficient in a bump-mapping matrix. The default value is 0.0.

D3DTSS_BUMPENVMAT10
The texture-stage state is a value for the [1][0] coefficient in a bump-mapping matrix. The default value is 0.0.

D3DTSS_BUMPENVMAT11
The texture-stage state is a value for the [1][1] coefficient in a bump-mapping matrix. The default value is 0.0.

D3DTSS_TEXCOORDINDEX
Index of the texture coordinate set to use with this texture stage. This flag is used only for fixed-function vertex processing. For example, it should not be used with vertex shaders. When rendering using vertex shaders, each stage's texture coordinate index must be set to its default value. The default index for each stage is equal to the stage index. Set this state to the zero-based index of the coordinate set for each vertex that this texture stage uses. You can specify up to eight sets of texture coordinates per vertex. If a vertex does not include a set of texture coordinates at the specified index, the system defaults to the u and v coordinates (0,0).

Additionally, applications can include, as logical **Or** with the index being set, one of the following flags to request that Microsoft® Direct3D® automatically generate the input texture coordinates for a texture transformation. With the

exception of D3DTSS_TCI_PASSTHRU, which resolves to zero, if any of the following values is included with the index being set, the system uses the index strictly to determine texture wrapping mode. These flags defined by the **CONST_D3DTSS_TCI_FLAGS** enumeration are most useful when performing environment mapping.

D3DTSS_TCI_PASSTHRU
Use the specified texture coordinates contained within the vertex format. This value resolves to zero.

D3DTSS_TCI_CAMERASPACENORMAL
Use the vertex normal, transformed to camera space, as the input texture coordinates for this stage's texture transformation.

D3DTSS_TCI_CAMERASPACEPOSITION
Use the vertex position, transformed to camera space, as the input texture coordinates for this stage's texture transformation.

D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR
Use the reflection vector, transformed to camera space, as the input texture coordinate for this stage's texture transformation. The reflection vector is computed from the input vertex position and normal vector.

D3DTSS_ADDRESSU
Member of the **CONST_D3DTEXTUREADDRESS** enumeration. Selects the texture-addressing method for the u coordinate. The default is D3DTADDRESS_WRAP.

D3DTSS_ADDRESSV
Member of the **CONST_D3DTEXTUREADDRESS** enumeration. Selects the texture-addressing method for the v coordinate. The default value is D3DTADDRESS_WRAP.

D3DTSS_BORDERCOLOR
Value that describes the color to use for rasterizing texture coordinates outside the [0.0,1.0] range. The default color is &H00000000.

D3DTSS_MAGFILTER
Member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration that indicates the texture magnification filter to use when rendering the texture onto primitives. The default value is D3DTEXF_POINT.

D3DTSS_MINFILTER
Member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration that indicates the texture minification filter to use when rendering the texture onto primitives. The default value is D3DTEXF_POINT.

D3DTSS_MIPFILTER
Member of the **CONST_D3DTEXTUREFILTERTYPE** enumeration that indicates the texture filter to use between mipmap levels. The default value is D3DTEXF_NONE.

D3DTSS_MIPMAPLODBIAS
Level of detail bias for mipmaps. Can be used to make textures appear more chunky or more blurred. The default value is 0.0.

Each unit of bias (+/-1.0) alters the selection by exactly one mipmap level. A negative bias causes the use of larger mipmap levels; the result is a sharper but more aliased image. A positive bias causes the use of smaller mipmap levels; the result is a more blurred image. A positive bias also causes less texture data to be referenced, which can boost performance on some systems.

D3DTSS_MAXMIPLEVEL
Maximum mipmap level of detail that the application allows, expressed as an index from the top of the mipmap chain. Lower values identify higher levels of detail within the mipmap chain. Zero, which is the default, indicates that all levels can be used. Nonzero values indicate that the application cannot display mipmaps that have a higher level of detail than the mipmap at the specified index.

D3DTSS_MAXANISOTROPY
Maximum level of anisotropy. The default value is 1.

D3DTSS_BUMPENVLSCALE
Scale for bump-map luminance. The default value is 0.0.

D3DTSS_BUMPENVLOFFSET
Offset for bump-map luminance. The default value is 0.0.

D3DTSS_TEXTURETRANSFORMFLAGS
Member of the **CONST_D3DTEXTURETRANSFORMFLAGS** enumeration that controls the transformation of texture coordinates for this texture stage. The default value is D3DTTFF_DISABLE.

D3DTSS_ADDRESSW
Member of the **CONST_D3DTEXTUREADDRESS** enumeration. Selects the texture-addressing method for the w coordinate. The default value is D3DTADDRESS_WRAP.

D3DTSS_COLORARG0
Settings for the third color operand for triadic operations (multiply, add, and linear interpolation), identified by one of the texture argument flags. This setting is supported if the D3DTEXOPCAPS_MULTIPLYADD or D3DTEXOPCAPS_LERP device capabilities are present.

Specify D3DTA_TEMP to select a temporary register color for read or write. D3DTA_TEMP is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present. The default value for the register is (0.0, 0.0, 0.0, 0.0).

D3DTSS_ALPHAARG0
Settings for the alpha channel selector operand for triadic operations (multiply add and linear interpolation), identified by one of the texture argument flags. This setting is supported if the D3DTEXOPCAPS_MULTIPLYADD or D3DTEXOPCAPS_LERP device capabilities are present.

Specify D3DTA_TEMP to select a temporary register color for read or write. D3DTA_TEMP is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present. The default value for the register is (0.0, 0.0, 0.0, 0.0)

D3DTSS_RESULTARG
Setting to select destination register for the result of this stage, identified by one of the texture argument flags. This value can be set to D3DTA_CURRENT (the

default value) or to D3DTA_TEMP, which is a single temporary register that can be read into subsequent stages as an input argument. The final color passed to the fog blender and frame buffer is taken from D3DTA_CURRENT, so the last active texture stage state must be set to write to current.

This setting is supported if the D3DPMISCCAPS_TSSARGTEMP device capability is present.

Remarks

Members of this enumeration are used with the **Direct3DDevice8.GetTextureStageState** and **Direct3DDevice8.SetTextureStageState** methods to retrieve and set texture state values.

The valid range of values for the D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT01, D3DTSS_BUMPENVMAT10, and D3DTSS_BUMPENVMAT11 bump-mapping matrix coefficients is greater than or equal to –8.0 and less than 8.0. This range, expressed in mathematical notation is [–8.0,8.0).

See Also

Direct3DDevice8.GetTextureStageState, **Direct3DDevice8.SetTextureStageState**

CONST_D3DTEXTURETRANSFORMFLAGS

#Defines texture-stage state values.

```
Enum CONST_D3DTEXTURETRANSFORMFLAGS
   D3DTTFF_DISABLE   =  0
   D3DTTFF_COUNT1    =  1
   D3DTTFF_COUNT2    =  2
   D3DTTFF_COUNT3    =  3
   D3DTTFF_COUNT4    =  4
   D3DTTFF_PROJECTED = 256 (&H100)
End Enum
```

Constants

D3DTTFF_DISABLE
Texture coordinates are passed directly to the rasterizer.

D3DTTFF_COUNT1
The rasterizer should expect 1-D texture coordinates.

IDH_CONST_D3DTEXTURETRANSFORMFLAGS_graphicsvb

D3DTTFF_COUNT2
The rasterizer should expect 2-D texture coordinates.

D3DTTFF_COUNT3
The rasterizer should expect 3-D texture coordinates.

D3DTTFF_COUNT4
The rasterizer should expect 4-D texture coordinates.

D3DTTFF_PROJECTED
The texture coordinates are all divided by the last element before being passed to the rasterizer. For example, if this flag is specified with the D3DTTFF_COUNT3 flag, the first and second texture coordinates is divided by the third coordinate before being passed to the rasterizer.

Remarks

These flags are used to set the value of the D3DTSS_TEXTURETRANSFORMFLAGS texture stage state for the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

See Also

CONST_D3DTEXTURESTAGESTATETYPE

CONST_D3DTRANSFORMSTATETYPE

#Defines constants that describe transformation state values.

```
Enum CONST_D3DTRANSFORMSTATETYPE
   D3DTS_WORLD     = 1
   D3DTS_VIEW      = 2
   D3DTS_PROJECTION = 3
   D3DTS_WORLD1    = 4
   D3DTS_WORLD2    = 5
   D3DTS_WORLD3    = 6
   D3DTS_TEXTURE0   = 16 (&H10)
   D3DTS_TEXTURE1   = 17 (&H11)
   D3DTS_TEXTURE2   = 18 (&H12)
   D3DTS_TEXTURE3   = 19 (&H13)
   D3DTS_TEXTURE4   = 20 (&H14)
   D3DTS_TEXTURE5   = 21 (&H15)
   D3DTS_TEXTURE6   = 22 (&H16)
   D3DTS_TEXTURE7   = 23 (&H17)
End Enum
```

IDH_CONST_D3DTRANSFORMSTATETYPE_graphicsvb

Constants

D3DTS_WORLD
Identifies the transformation matrix being set as the world transformation matrix. The default value is Nothing (the identity matrix).

D3DTS_VIEW
Identifies the transformation matrix being set as the view transformation matrix. The default value is NULL (the identity matrix).

D3DTS_PROJECTION
Identifies the transformation matrix being set as the projection transformation matrix. The default value is Nothing (the identity matrix).

D3DTS_WORLD1 through D3DTS_WORLD3
Identifies subsequent transformation matrices that can be used to blend vertices by using the corresponding matrix and a blending (beta) weight value specified in the vertex format.

D3DTS_TEXTURE0 through D3DTS_TEXTURE7
Identifies the transformation matrix being set for the specified texture stage.

See Also

Direct3DDevice8.GetTransform, **Direct3DDevice8.MultiplyTransform**, **Direct3DDevice8.SetTransform**

CONST_D3DTSS_TCI_FLAGS

#Defines constants specifying that Microsoft® Direct3D® automatically generate the input texture coordinates for a texture transformation.

```
Enum CONST_D3DTSS_TCI_FLAGS
   D3DTSS_TCI_PASSTHRU              =     0
   D3DTSS_TCI_CAMERASPACENORMAL        =  65536 (&H10000)
   D3DTSS_TCI_CAMERASPACEPOSITION      = 131072 (&H20000)
   D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR = 196608
(&H30000)
End Enum
```

Constants

D3DTSS_TCI_PASSTHRU
Use the specified texture coordinates contained within the vertex format. This value resolves to zero.

D3DTSS_TCI_CAMERASPACENORMAL

IDH_CONST_D3DTSS_TCI_FLAGS_graphicsvb

Use the vertex normal, transformed to camera space, as the input texture coordinates for this stage's texture transformation.

D3DTSS_TCI_CAMERASPACEPOSITION
Use the vertex position, transformed to camera space, as the input texture coordinates for this stage's texture transformation.

D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR
Use the reflection vector, transformed to camera space, as the input texture coordinate for this stage's texture transformation. The reflection vector is computed from the input vertex position and normal vector.

Remarks

These flags are used to set the value of the D3DTSS_TEXCOORDINDEX member for the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

See Also

CONST_D3DTEXTURESTAGESTATETYPE

CONST_D3DUSAGEFLAGS

#Defines constants that define usages for resources.

```
Enum CONST_D3DUSAGEFLAGS
   D3DUSAGE_RENDERTARGET      =   1
   D3DUSAGE_DEPTHSTENCIL      =   2
   D3DUSAGE_WRITEONLY         =   8
   D3DUSAGE_SOFTWAREPROCESSING =  16 (&H10)
   D3DUSAGE_DONOTCLIP         =  32 (&H20)
   D3DUSAGE_POINTS            =  64 (&H40)
   D3DUSAGE_RTPATCHES         = 128 (&H80)
   D3DUSAGE_NPATCHES          = 256 (&H100)
   D3DUSAGE_DYNAMIC           = 512 (&H200)
End Enum
```

Constants

D3DUSAGE_RENDERTARGET
Set to indicate that the surface is to be used as a render target. The resource can be passed to the *RenderTarget* parameter of the **Direct3DDevice8.SetRenderTarget** method.

This usage is automatically applied for a render target resource.

D3DUSAGE_DEPTHSTENCIL

IDH_CONST_D3DUSAGEFLAGS_graphicsvb

Set to indicate that the surface is to be used as a depth-stencil surface. The resource can be passed to the *NewZStencil* parameter of **SetRenderTarget**.

This usage is automatically applied for a depth-stencil resource.

D3DUSAGE_WRITEONLY
Informs the system that the application writes only to the buffer. Using this flag enables the driver to select the best memory location for efficient write operations and rendering. Attempts to read from a buffer that is created with this capability can result in degraded performance.

This flag is valid only for index and vertex buffer resources.

D3DUSAGE_SOFTWAREPROCESSING
Set to indicate that the buffer is to be used with software processing.

This flag is valid only for index and vertex buffer resources.

D3DUSAGE_DONOTCLIP
Set to indicate that the vertex buffer content will never require clipping.

This flag is valid only for vertex buffer resources.

D3DUSAGE_POINTS
Set to indicate when a vertex or index buffer is to be used for drawing point sprites or indexed point lists.

D3DUSAGE_RTPATCHES
Set to indicate when a vertex or index buffer is to be used for drawing high-order surfaces.

D3DUSAGE_NPATCHES
Set to indicate when the vertex or index buffer is to be used for drawing N patches.

D3DUSAGE_DYNAMIC
Set to indicate when the vertex or index buffer requires dynamic memory usage. This usage is useful for drivers since it enables them to decide where to place the driver. In general, static vertex buffers are placed in video memory and dynamic vertex buffers are placed in AGP memory. Note that there is no separate static usage. If you do not specify D3DUSAGE_DYNAMIC, the vertex buffer is made static. D3DUSAGE_DYNAMIC is strictly enforced through the D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE locking flags. As a result, D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE are valid only on vertex and index buffers created with D3DUSAGE_DYNAMIC. They are not valid flags on static vertex buffers.

Note that D3DUSAGE_DYNAMIC cannot be specified on managed vertex and index buffers. For more information, see Managing Resources.

This flag is valid only for index and vertex buffer resources.

See Also

Direct3DDevice8.CreateCubeTexture, **Direct3DDevice8.CreateIndexBuffer**, **Direct3DDevice8.CreateTexture**, **Direct3DDevice8.CreateVertexBuffer**, **Direct3DDevice8.CreateVolumeTexture**

CONST_D3DVERTEXBLENDFLAGS

#Defines flags used to control the number or matrices that the system applies when performing multimatrix vertex blending.

```
Enum CONST_D3DVERTEXBLENDFLAGS
   D3DVBF_DISABLE    =  0
   D3DVBF_1WEIGHTS   =  1
   D3DVBF_2WEIGHTS   =  2
   D3DVBF_3WEIGHTS   =  3
   D3DVBF_TWEENING   = 255 (&HFF)
   D3DVBF_0WEIGHTS   = 256 (&H100)
End Enum
```

Constants

D3DVBF_DISABLE
: Disable vertex blending; apply only the world matrix specified by the D3DTS_WORLD transformation state.

D3DVBF_1WEIGHTS
: Enable vertex blending between the two matrices set by the D3DTS_WORLD and D3DTS_WORLD1 transformation states.

D3DVBF_2WEIGHTS
: Enable vertex blending between the three matrices set by the D3DTS_WORLD, D3DTS_WORLD1, and D3DTS_WORLD2 transformation states.

D3DVBF_3WEIGHTS
: Enable vertex blending between the four matrices set by the D3DTS_WORLD, D3DTS_WORLD1, D3DTS_WORLD2, and D3DTS_WORLD3 transformation states.

D3DVBF_TWEENING
: Vertex blending is done by using the value assigned to D3DRS_TWEENFACTOR.

D3DVBF_0WEIGHTS
: Use a single matrix with a weight of 1.0.

Remarks

These flags are used to set the value of the D3DRS_VERTEXBLEND render state for the **CONST_D3DRENDERSTATETYPE** enumeration.

Geometry blending (multimatrix vertex blending) requires that your application use a vertex format that has blending (beta) weights for each vertex.

IDH_CONST_D3DVERTEXBLENDFLAGS_graphicsvb

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DVTXPCAPSFLAGS

#Defines vertex processing capabilities.

```
Enum CONST_D3DVTXPCAPSFLAGS
   D3DVTXPCAPS_TEXGEN          =  1
   D3DVTXPCAPS_MATERIALSOURCE7   =  2
   D3DVTXPCAPS_DIRECTIONALLIGHTS =  8
   D3DVTXPCAPS_POSITIONALLIGHTS  = 16 (&H10)
   D3DVTXPCAPS_LOCALVIEWER       = 32 (&H20)
   D3DVTXPCAPS_TWEENING          = 64 (&H40)
End Enum
```

Constants

D3DVTXPCAPS_TEXGEN
Device can generate texture coordinates.

D3DVTXPCAPS_MATERIALSOURCE7
Device supports selectable vertex color sources.

D3DVTXPCAPS_DIRECTIONALLIGHTS
Device supports directional lights.

D3DVTXPCAPS_POSITIONALLIGHTS
Device supports positional lights (including point lights and spotlights).

D3DVTXPCAPS_LOCALVIEWER
Device supports local viewer.

D3DVTXPCAPS_TWEENING
Device supports vertex tweening.

Remarks

These flags can be combined and present in the **VertexProcessingCaps** member of **D3DCAPS8**.

See Also

D3DCAPS8

IDH_CONST_D3DVTXPCAPSFLAGS_graphicsvb

CONST_D3DWRAPBIAS

#Defined to enable or disable texture wrapping based on the zero-based integer of a texture coordinate set.

```
Enum CONST_D3DWRAPBIAS
   D3DRENDERSTATE_WRAPBIAS = 128 (&H80)
End Enum
```

Constants

D3DRENDERSTATE_WRAPBIAS
Wrap bias. For implementation details, see Remarks.

Remarks

Microsoft® Direct3D® defines the D3DRENDERSTATE_WRAPBIAS constant as a convenience for applications to enable or disable texture wrapping based on the zero-based integer of a texture coordinate set (rather than explicitly using one of the D3DRS_WRAP*n* state values). Add the D3DRENDERSTATE_WRAPBIAS value to the zero-based index of a texture coordinate set to calculate the D3DRS_WRAP*n* value that corresponds to that index, as shown in the following example:

```
On Local Error Resume Next

' Enable U/V wrapping for textures that use the texture
' coordinate set at the index within the lIndex variable.
Call d3dDevice8.SetRenderState( _
      lIndex + D3DRENDERSTATE_WRAPBIAS, _
      D3DWRAPCOORD_0 Or D3DWRAPCOORD_1)

' If lIndex is 3, the value that results from
' the addition equates to D3DRENDERSTATE_WRAP3 (131).
If Err.Number <> DD_OK Then
  ' Code to handle error goes here.
End If
```

See Also

CONST_D3DRENDERSTATETYPE

IDH_CONST_D3DWRAPBIAS_graphicsvb

CONST_D3DWRAPFLAGS

#Defines texture wrapping behavior for multiple sets of texture coordinates.

```
Enum CONST_D3DWRAPFLAGS
   D3DWRAP_U      = 1
   D3DWRAPCOORD_0 = 1
   D3DWRAP_V      = 2
   D3DWRAPCOORD_1 = 2
   D3DWRAP_W      = 4
   D3DWRAPCOORD_2 = 4
   D3DWRAPCOORD_3 = 8
End Enum
```

Constants

D3DWRAP_U and D3DWRAPCOORD_0
: Wrap in the direction of the first dimension for a given texture.

D3DWRAP_V and D3DWRAPCOORD_1
: Wrap in the direction of the second dimension for a given texture.

D3DWRAP_W and D3DWRAPCOORD_2
: Wrap in the direction of the third dimension for a given texture.

D3DWRAPCOORD_3
: Wrap in the direction of the fourth dimension for a given texture.

Remarks

These flags set the value of the D3DRS_WRAP0 through D3DRS_WRAP7 render states for the **CONST_D3DRENDERSTATETYPE** enumeration.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DZBUFFERTYPE

#Defines constants that describe depth-buffer formats.

```
Enum CONST_D3DZBUFFERTYPE
   D3DZB_FALSE = 0
   D3DZB_TRUE  = 1
   D3DZB_USEW  = 2
End Enum
```

IDH_CONST_D3DWRAPFLAGS_graphicsvb
IDH_CONST_D3DZBUFFERTYPE_graphicsvb

Constants

D3DZB_FALSE
Disable depth buffering.

D3DZB_TRUE
Enable z-buffering.

D3DZB_USEW
Enable w-buffering.

Remarks

These flags are used to set the value of the D3DRS_ZENABLE render state for the **CONST_D3DRENDERSTATETYPE** enumeration.

See Also

CONST_D3DRENDERSTATETYPE

Texture Argument Flags

Each texture stage for a device can have two texture arguments that affect the color or alpha channel of the texture.

Set and retrieve texture arguments by calling the **Direct3DDevice8.SetTextureStageState** and **Direct3DDevice8.GetTextureStageState**, and specifying the D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAARG1 or D3DTSS_ALPHAARG2 constants of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

The following flags, organized as arguments and modifiers, can be used with color and alpha arguments for a texture stage. You can combine an argument flag with a modifier, but you cannot combine two argument flags.

Argument flags
D3DTA_CURRENT
The texture argument is the result of the previous blending stage. In the first texture stage (stage 0), this argument is equivalent to D3DTA_DIFFUSE. If the previous blending stage uses a bump-map texture (the D3DTOP_BUMPENVMAP operation), the system chooses the texture from the stage before the bump-map texture. If *s* represents the current texture stage, and *s* - 1 contains a bump-map texture, this argument becomes the result output by texture stage *s* - 2. Permissions are read/write.

D3DTA_DIFFUSE

The texture argument is the diffuse color interpolated from vertex components during Gouraud shading. If the vertex does not contain a diffuse color, the default color is &HFFFFFFFF. Permissions are read-only.

D3DTA_SELECTMASK
Mask value for all arguments; not used when setting texture arguments.

D3DTA_SPECULAR
The texture argument is the specular color interpolated from vertex components during Gouraud shading. If the vertex does not contain a specular color, the default color is &HFFFFFFFF. Permissions are read only.

D3DTA_TEMP
Select the temporary register color. Permissions are read/write.

D3DTA_TEXTURE
The texture argument is the texture color for this texture stage. If no texture is set for a stage that uses this blending argument, the system defaults to a color value of R: 1.0, G: 1.0, B: 1.0 for color, and 1.0 for alpha. Permissions are read-only.

D3DTA_TFACTOR
The texture argument is the texture factor set in a previous call to the **Direct3DDevice8.SetRenderState** with the D3DRS_TEXTUREFACTOR render state value.

Modifier flags

D3DTA_ALPHAREPLICATE
Replicate the alpha information to all color channels before the operation completes.

D3DTA_COMPLEMENT
Invert the argument so that, if the result of the argument were referred to by the variable *x*, the value would be 1.0 - *x*.

Flexible Vertex Format Flags

The flexible vertex format (FVF) is used to describe the contents of vertices stored interleaved in a single data stream. A FVF code is generally used to specify data to be processed by fixed function vertex processing.

The following flags defined by the **CONST_D3DFVFFLAGS** enumeration, describe a vertex format.

Flexible vertex format (FVF) flags

D3DFVF_DIFFUSE
Vertex format includes a diffuse color component.

D3DFVF_NORMAL
Vertex format includes a vertex normal vector. This flag cannot be used with the D3DFVF_XYZRHW flag.

D3DFVF_PSIZE
Vertex format specified in point size. This size is expressed in camera space units for vertices that are not transformed and lit, and in device-space units for transformed and lit vertices.

D3DFVF_SPECULAR
Vertex format includes a specular color component.

D3DFVF_VERTEX)
Same as specifying (D3DFVF_XYZ Or D3DFVF_NORMAL Or D3DFVF_TEX1).

D3DFVF_XYZ
Vertex format includes the position of an untransformed vertex. This flag cannot be used with the D3DFVF_XYZRHW flag.

D3DFVF_XYZRHW
Vertex format includes the position of a transformed vertex. This flag cannot be used with the D3DFVF_XYZ or D3DFVF_NORMAL flags.

D3DFVF_XYZB1 through D3DFVF_XYZB5
Vertex format contains position data, and a corresponding number of weighting (beta) values to use for multimatrix vertex blending operations. Currently, Microsoft® Direct3D® can blend with up to three weighting values and four blending matrices.

Texture-related FVF flags

D3DFVF_TEX0 through D3DFVF_TEX8
Number of texture coordinate sets for this vertex. The actual values for these flags are not sequential. For information on additional texture-related FVF flags, see the **CONST_D3DFVFTEXTUREFORMATS** enumeration.

Mask values

D3DFVF_POSITION_MASK
Mask for position bits.

D3DFVF_RESERVED0 and D3DFVF_RESERVED2
Mask values for reserved bits in the flexible vertex format. Do not use.

D3DFVF_TEXCOUNT_MASK
Mask value for texture flag bits.

Miscellaneous

D3DFVF_TEXCOUNT_SHIFT
The number of bits by which to shift an integer value that identifies the number of a texture coordinates for a vertex.

See Also

About Vertex Formats, Geometry Blending

Four-Character Codes (FOURCC)

Microsoft® Direct3D® and the Direct3DX utility library use a special set of codes that are four characters in length. These codes, called four-character codes or FOURCCs, are stored in file headers of files that contain multimedia data, such as bitmap images, sound, or video. FOURCCs describe the software technology that was used to produce multimedia data. By implication, they also describe the format of the data itself.

Direct3D applications use FOURCCs for image color and format conversion.

FOURCCs are registered with Microsoft by the vendors of the respective multimedia software technologies. Some common FOURCCs appear in the following list.

FOURCC	Company	Technology name
AUR2	AuraVision Corporation	AuraVision Aura 2: YUV 422
AURA	AuraVision Corporation	AuraVision Aura 1: YUV 411
CHAM	Winnov, Inc.	MM_WINNOV_CAVIARA_CHAMPAGNE
CVID	Supermac	Cinepak by Supermac
CYUV	Creative Labs, Inc.	Creative Labs YUV
DXT1	Microsoft Corporation	DirectX Texture Compression Format 1
DXT2	Microsoft Corporation	DirectX Texture Compression Format 2
DXT3	Microsoft Corporation	DirectX Texture Compression Format 3
DXT4	Microsoft Corporation	DirectX Texture Compression Format 4
DXT5	Microsoft Corporation	DirectX Texture Compression Format 5
FVF1	Iterated Systems, Inc.	Fractal Video Frame
IF09	Intel Corporation	Intel Intermediate YUV9
IV31	Intel Corporation	Indeo 3.1
JPEG	Microsoft Corporation	Still Image JPEG DIB
MJPG	Microsoft Corporation	Motion JPEG DIB Format
MRLE	Microsoft Corporation	Run Length Encoding
MSVC	Microsoft Corporation	Video 1
PHMO	IBM Corporation	Photomotion
RT21	Intel Corporation	Indeo 2.1
ULTI	IBM Corporation	Ultimotion
V422	Vitec Multimedia	24-bit YUV 4:2:2
V655	Vitec Multimedia	16-bit YUV 4:2:2
VDCT	Vitec Multimedia	Video Maker Pro DIB
VIDS	Vitec Multimedia	YUV 4:2:2 CCIR 601 for V422
YU92	Intel Corporation	YUV
YUV8	Winnov, Inc.	MM_WINNOV_CAVIAR_YUV8
YUV9	Intel Corporation	YUV9
YUYV	Canopus, Co., Ltd.	BI_YUYV, Canopus
ZPEG	Metheus	Video Zipper

Error Codes

Errors are represented by negative values and cannot be combined. The following is a list of the values that can be returned by Microsoft® Direct3D® methods. See the individual method descriptions for lists of the values that each can return.

D3D_OK
: No error occurred.

D3DERR_CANNOTTATTRSORT
: Attribute sort (D3DXMESHOPT_ATTRSORT) is not supported as an optimization technique.

D3DERR_CONFLICTINGRENDERSTATE
: The currently set render states cannot be used together.

D3DERR_CONFLICTINGTEXTUREFILTER
: The current texture filters cannot be used together.

D3DERR_CONFLICTINGTEXTUREPALETTE
: The current textures cannot be used simultaneously. This generally occurs when a multitexture device requires that all palletized textures simultaneously enabled also share the same palette.

D3DERR_DEVICELOST
: The device is lost and cannot be restored at the current time, so rendering is not possible.

D3DERR_DEVICENOTRESET
: The device cannot be reset.

D3DERR_DRIVERINTERNALERROR
: Internal driver error.

D3DERR_INVALIDCALL
: The method call is invalid. For example, a method's parameter may have an invalid value.

D3DERR_INVALIDDEVICE
: The requested device type is not valid.

D3DERR_MOREDATA
: There is more data available than the specified buffer size can hold.

D3DERR_NOTAVAILABLE
: The queried technique is not supported by this device.

D3DERR_NOTFOUND
: The requested item was not found.

D3DERR_OUTOFVIDEOMEMORY
: Direct3D does not have enough display memory to perform the operation.

D3DERR_TOOMANYOPERATIONS
: The application is requesting more texture-filtering operations than the device supports.

D3DERR_UNSUPPORTEDALPHAARG

The device does not support a specified texture-blending argument for the alpha channel.

D3DERR_UNSUPPORTEDALPHAOPERATION
The device does not support a specified texture-blending operations for the alpha channel.

D3DERR_UNSUPPORTEDCOLORARG
The device does not support a specified texture-blending arguments for color values.

D3DERR_UNSUPPORTEDCOLOROPERATION
The device does not support a specified texture-blending operation for color values.

D3DERR_UNSUPPORTEDFACTORVALUE
The specified texture factor value is not supported by the device.

D3DERR_UNSUPPORTEDTEXTUREFILTER
The specified texture filter is not supported by the device.

D3DERR_WRONGTEXTUREFORMAT
The pixel format of the texture surface is not valid.

E_FAIL
An undetermined error occurred inside the Direct3D subsystem.

E_INVALIDARG
An invalid parameter was passed to the returning function.

E_INVALIDCALL
The method call is invalid. For example, a method's parameter may have an invalid value.

E_OUTOFMEMORY
Direct3D could not allocate sufficient memory to complete the call.

Direct3DX Visual Basic Reference

This section contains reference information for the API elements provided by the Direct3DX utility library. Reference material is divided into the following categories.

- Classes
- Functions
- Types
- Enumerations
- Error Codes

Classes

This section contains reference information for the classes provided by the Direct3DX utility library. The following classes are used with the Direct3DX utility library.

- **D3DX8**
- **D3DXBaseMesh**
- **D3DXBuffer**
- **D3DXFont**
- **D3DXMesh**
- **D3DXPMesh**
- **D3DXRenderToSurface**
- **D3DXSkinMesh**
- **D3DXSPMesh**
- **D3DXSprite**

D3DX8

#Applications use the methods of the **D3DX8** class to simplify complicated tasks such as creating textures, manipulating meshes, drawing shapes, and assembling shaders.

An instance of the **D3DX8** class is obtained by declaring an object variable and setting that variable to a new instance of **D3DX8**, as shown in the following code fragment.

```
Dim d3dx As D3DX8
Set d3dx = New D3DX8
```

The methods of the **D3DX8** class can be organized into the following groups.

Meshes	**BoxBoundProbe**
	CleanMesh
	ComputeBoundingBox
	ComputeBoundingBoxFromMesh
	ComputeBoundingSphere
	ComputeBoundingSphereFromMesh
	ComputeNormals
	CreateBuffer

IDH_D3DX8_graphicsd3dxvb

	CreateMesh
	CreateMeshFVF
	CreatePolygon
	CreateSkinMesh
	CreateSkinMeshFromMesh
	CreateSkinMeshFVF
	CreateSPMesh
	DeclaratorFromFVF
	FVFFromDeclarator
	GeneratePMesh
	Intersect
	LoadPMeshFromFile
	LoadSkinMeshFromXof
	LoadMeshFromX
	LoadMeshFromXof
	SaveMeshToX
	SavePMeshToFile
	SimplifyMesh
	SphereBoundProbe
	TesselateMesh
	ValidMesh
Miscellaneous	**BufferGetBoneCombo**
	BufferGetBoneComboBoneIds
	BufferGetBoneName
	BufferGetData
	BufferGetMaterial
	BufferGetTextureName
	BufferSetData
	CreateFont
	CreateRenderToSurface
	CreateSprite

	GetErrorString
	GetFVFVertexSize
Shaders	**AssembleShader**
	AssembleShaderFromFile
Shapes	**CreateBox**
	CreateCylinder
	CreateSphere
	CreateTeapot
	CreateText
	CreateTorus
Textures	**CheckCubeTextureRequirements**
	CheckTextureRequirements
	CheckVolumeTextureRequirements
	CreateCubeTexture
	CreateCubeTextureFromFile
	CreateCubeTextureFromFileEx
	CreateCubeTextureFromFileInMemory
	CreateCubeTextureFromFileInMemoryEx
	CreateTexture
	CreateTextureFromFile
	CreateTextureFromFileEx
	CreateTextureFromFileInMemory
	CreateTextureFromFileInMemoryEx
	CreateTextureFromResource
	CreateTextureFromResourceEx
	CreateVolumeTexture
	FilterCubeTexture
	FilterTexture
	FilterVolumeTexture
	LoadSurfaceFromFile

LoadSurfaceFromFileInMemory

LoadSurfaceFromMemory

LoadSurfaceFromResource

LoadSurfaceFromSurface

LoadVolumeFromMemory

LoadVolumeFromVolume

D3DX8.AssembleShader

#Assembles an ASCII description of a pixel or vertex shader into binary form, where the shader source is in memory.

object.**AssembleShader(** _
SrcData **As Any**, _
Flags **As Long**, _
Constants **As D3DXBuffer) As D3DXBuffer**

Parts

object
Object expression that resolves to a **D3DX8** object.

SrcData
Source code from which to assemble the shader.

Flags
A combination of the members of the **CONST_D3DXASM** type, specifying assembly options.

Constants
D3DXBuffer object, to be filled with the constant declarations. These constants are returned as a vertex shader declaration fragment. It is up to the application to insert the contents of this buffer into their declaration. For pixel shaders this parameter is meaningless because constant declarations are included in the assembled shader. This parameter is ignored if it is NULL.

Return Values

Returns a **D3DXBuffer** object, containing the compiled object code.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

IDH_D3DX8.AssembleShader_graphicsd3dxvb

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.AssembleShaderFromFile

#Assembles an ASCII description of a pixel or vertex shader into binary form.

object.**AssembleShaderFromFile(** _
SrcFile **As String**, _
Flags **As Long**, _
ErrLog **As String**, _
Constants **As D3DXBuffer) As D3DXBuffer**

Parts

object
Object expression that resolves to a **D3DX8** object.

SrcFile
Source file name.

Flags
A combination of the members of the **CONST_D3DXASM** type, specifying assembly options.

ErrLog
Contains ASCII error messages.

Constants
D3DXBuffer object, to be filled with the constant declarations. These constants are returned as a vertex shader declaration fragment. It is up to the application to insert the contents of this buffer into their declaration. For pixel shaders this parameter is meaningless because constant declarations are included in the assembled shader. This parameter is ignored if it is NULL.

Return Values

Returns a **D3DXBuffer** object, containing the compiled object code.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

IDH_D3DX8.AssembleShaderFromFile_graphicsd3dxvb

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.BoxBoundProbe

#Determines if a ray intersects the volume of a box's bounding box.

object.**BoxBoundProbe(** _
MinVert **As D3DVECTOR**, _
MaxVert **As D3DVECTOR**, _
RayPosition **As D3DVECTOR**, _
Raydirection **As D3DVECTOR) As Boolean**

Parts

object
Object expression that resolves to a **D3DX8** object.

MinVert
A **D3DVECTOR** type, describing the lower-left corner of the bounding box. See Remarks.

MaxVert
A **D3DXVECTOR** type, describing the upper-right corner of the bounding box. See Remarks.

RayPosition
A **D3DXVECTOR** type, specifying the origin coordinate of the ray.

Raydirection
A **D3DXVECTOR** type, specifying the direction of the ray.

Return Values

Returns TRUE if the ray intersects the volume of the box's bounding box. Otherwise, returns FALSE.

Remarks

D3DX8.BoxBoundProbe determines if the ray intersects the volume of the box's bounding box, not just the surface of the box.

IDH_D3DX8.BoxBoundProbe_graphicsd3dxvb

The **D3DXVECTOR** values passed to **D3DX8.BoxBoundProbe** are xmin, xmax, ymin, ymax, zmin, and zmax. Thus, the following defines the corners of the bounding box.

```
xmax, ymax, zmax
xmax, ymax, zmin
xmax, ymin, zmax
xmax, ymin, zmin
xmin, ymax, zmax
xmin, ymax, zmin
xmin, ymin, zmax
xmin, ymin, zmin
```

The depth of the bounding box in the z direction is zmax - zmin, in the y direction is ymax - ymin, and in the x direction is xmax - xmin. For example, with the following minimum and maximum vectors, min (-1, -1, -1) and max (1, 1, 1), the bounding box is defined in the following manner.

```
 1,  1,  1
 1,  1, -1
 1, -1,  1
 1, -1, -1
-1,  1,  1
-1,  1, -1
-1, -1,  1
-1, -1, -1
```

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.BufferGetBoneCombo

#Retrieves a bone combination from a buffer.

```
object.BufferGetBoneCombo( _
    BoneComboBuffer As D3DXBuffer, _
    Index As Long, _
    BoneCombo As D3DXBONECOMBINATION)
```

IDH_D3DX8.BufferGetBoneCombo_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DX8** object.

BoneComboBuffer
D3DXBuffer object, the buffer containing the bone combination data.

Index
Index value, specifying the index into the buffer.

BoneCombo
D3DXBONECOMBINATION type, to be filled with the bone combination data for the bone specified by *Index*.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DX8.LoadSkinMeshFromXof, **D3DXSkinMesh.ConvertToBlendedMesh**

D3DX8.BufferGetBoneComboBoneIds

Retrieves the bone IDs from a bone combination buffer.

```
#object.BufferGetBoneComboBoneIds( _
    BoneComboBuffer As D3DXBuffer, _
    Index As Long, _
    PaletteSize As Long, _
    BoneIds As Any)
```

Parts

object
Object expression that resolves to a **D3DX8** object.

BoneComboBuffer
D3DXBuffer object, the buffer containing the bone data.

Index
Index value, specifying the index into the buffer.

PaletteSize
The maximum number of matrices that the bone combination can have.

IDH_D3DX8.BufferGetBoneComboBoneIds_graphicsd3dxvb

BoneIds
First element of an array of values that identify each of the bones that can be drawn in a single drawing call. Note that the array can be of variable length to accommodate variable length bone combinations of **D3DXSkinMesh.ConvertToIndexedBlendedMesh**.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXSkinMesh.GetMaxVertexInfluences, D3DXSkinMesh.GetNumBoneInfluences

D3DX8.BufferGetBoneName

#Retrieves the name of a bone from a buffer.

object.**BufferGetBoneName(** _
BoneNameBuffer **As D3DXBuffer**, _
Index **As Long) As String**

Parts

object
Object expression that resolves to a **D3DX8** object.

BoneNameBuffer
D3DXBuffer object, the buffer containing the bone data.

Index
Index value, specifying the index into the buffer.

Return Values

String identifying the name of the bone.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

IDH_D3DX8.BufferGetBoneName_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.BufferGetData

#Retrieves data from a buffer.

object.**BufferGetData(** _
Buffer **As D3DXBuffer**, _
Index **As Long**, _
TypeSize **As Long**, _
TypeCount **As Long**, _
Data **As Any)**

Parts

object
Object expression that resolves to a **D3DX8** object.

Buffer
D3DXBuffer object, the buffer containing the data to retrieve.

Index
Index value, specifying the index into the buffer.

TypeSize
Size of the data type to retrieve from the buffer, in bytes.

TypeCount
Count of elements to retrieve from the buffer.

Data
After the method returns, an object holding the retrieved buffer data.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.BufferGetMaterial

#Retrieves material from a buffer.

object.**BufferGetMaterial(** _
MaterialBuffer **As D3DXBuffer**, _

IDH_D3DX8.BufferGetData_graphicsd3dxvb
IDH_D3DX8.BufferGetMaterial_graphicsd3dxvb

Index **As Long, _**
Mat **As D3DMATERIAL8)**

Parts

object
Object expression that resolves to a **D3DX8** object.

MaterialBuffer
D3DXBuffer object, the buffer containing the material to retrieve.

Index
Index value, specifying the buffer's index.

Mat
D3DMATERIAL8 object; the material retrieved from the queried buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.BufferGetTextureName

#Retrieves the name of a texture from a buffer.

object.**BufferGetTextureName(_**
MaterialBuffer **As D3DXBuffer**, **_**
Index **As Long) As String**

Parts

object
Object expression that resolves to a **D3DX8** object.

MaterialBuffer
D3DXBuffer object, the buffer containing the material to retrieve.

Index
Index value, specifying the buffer's index.

Return Values

Returned **String**, identifying the name of the texture contained in the queried buffer.

IDH_D3DX8.BufferGetTextureName_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.BufferSetData

#Sets data in a buffer.

object.**BufferSetData(** _
Buffer **As D3DXBuffer**, _
Index **As Long**, _
TypeSize **As Long**, _
TypeCount **As Long**, _
Data **As Any)**

Parts

object
Object expression that resolves to a **D3DX8** object.

Buffer
D3DXBuffer object, the buffer to load with data.

Index
Index value, specifying the index into the buffer.

TypeSize
Size of the data type to load into the buffer, in bytes.

TypeCount
Count of elements to load into the buffer.

Data
An object holding the data to load into the buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DX8.BufferSetData_graphicsd3dxvb

D3DX8.CheckCubeTextureRequirements

[#]Checks cube texture creation parameters.

object.**CheckCubeTextureRequirements(** _
Device **As Direct3DDevice8**, _
Size **As Long**, _
NumMipLevels **As Long**, _
Usage **As Long**, _
Format **As CONST_D3DFORMAT**, _
Pool As **CONST_D3DPOOL)**

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device associated with the cube texture.

Size
Requested width and height in pixels, or ByVal 0. Returns the corrected size.

NumMipLevels
Number of requested mipmap levels, or ByVal 0. Returns the corrected number of mipmap levels.

Usage
0 or the D3DUSAGE_RENDERTARGET member of the **CONST_D3DUSAGEFLAGS** enumeration. Specifying D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **Direct3D8.CheckDeviceFormat**.

Format
Member of the **CONST_D3DFORMAT** enumeration. Specifies the desired pixel format. Returns the corrected format.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the cube texture should be placed.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

IDH_D3DX8.CheckCubeTextureRequirements_graphicsd3dxvb

D3DERR_NOTAVAILABLE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **Direct3DCubeTexture8.GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

If parameters to this method are invalid, this function returns corrected parameters.

D3DX8.CheckTextureRequirements

#Checks texture creation parameters.

object.**CheckTextureRequirements(** _
Device **As Direct3DDevice8**, _
Width **As Long**, _
Height **As Long**, _
NumMipLevels **As Long**, _
Usage **As Long**, _
Format **As CONST_D3DFORMAT**, _
Pool As **CONST_D3DPOOL)**

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device associated with the cube texture.

Width
Requested width in pixels, or ByVal 0. Returns the corrected size.

Height
Requested height in pixels, or ByVal 0. Returns the corrected size.

NumMipLevels
Number of requested mipmap levels, or ByVal 0. Returns the corrected number of mipmap levels.

Usage
0 or the D3DUSAGE_RENDERTARGET member of the **CONST_D3DUSAGEFLAGS** enumeration. Specifying D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a

IDH_D3DX8.CheckTextureRequirements_graphicsd3dxvb

render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **Direct3D8.CheckDeviceFormat**.

Format

Member of the **CONST_D3DFORMAT** enumeration. Specifies the desired pixel format. Returns the corrected format.

Pool

Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the texture should be placed.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If parameters to this method are invalid, this function returns corrected parameters.

D3DX8.CheckVolumeTextureRequirements

#Checks volume texture creation parameters.

object.**CheckVolumeTextureRequirements(** _
Device **As Direct3DDevice8**, _
Width **As Long**, _
Height **As Long**, _
Depth **As Long**, _
NumMipLevels **As Long**, _
Usage **As Long**, _
PixelFormat **CONST_D3DFORMAT**, _
Pool **As CONST_D3DPOOL)**

Parts

object

IDH_D3DX8.CheckVolumeTextureRequirements_graphicsd3dxvb

Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device associated with the volume texture.

Width
Requested width in pixels, or ByVal 0. Returns the corrected size.

Height
Requested height in pixels, or ByVal 0. Returns the corrected size.

Depth
Requested depth in pixels, or ByVal 0. Returns the corrected size.

NumMipLevels
Number of requested mipmap levels, or ByVal 0. Returns the corrected number of mipmap levels.

Usage
Currently not used, set to 0.

PixelFormat
Member of the **CONST_D3DFORMAT** enumeration. Specifies the desired pixel format. Returns the corrected format.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the volume texture should be placed.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If parameters to this function are invalid, this function returns corrected parameters.

D3DX8.CleanMesh

#Cleans a mesh, preparing it for simplification.

object.**CleanMesh(** _
MeshIn **As D3DXMesh**, _

IDH_D3DX8.CleanMesh_graphicsd3dxvb

Adjacency **As Any) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

MeshIn
D3DXMesh object, representing the mesh to be cleaned.

Adjacency
First element of an array of three **Long** values per face that specify the three neighbors for each face in the mesh to be cleaned. The same mesh is returned that was passed in if no cleaning was necessary.

Return Values

D3DXMesh object, representing the returned cleaned mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method cleans the mesh by adding another vertex where two fans of triangles share the same vertex.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim d3dxbAdjacency As D3DXBuffer
Dim mesh As D3DXMesh

' This code fragment assumes that d3dxbAdjacency and mesh have been properly
' initialized.
Call D3DX8.CleanMesh(d3dxm, ByVal d3dxbAdjacency.GetBufferPointer)
```

D3DX8.ComputeBoundingBox

#Computes a bounding box.

object.**ComputeBoundingBox(** _
PointsFVF **As Any**, _
NumVertices **As Long**, _
FVF **As Long**, _
MinArray **As D3DVECTOR**, _
MaxArray **As D3DVECTOR)**

Parts

object
Object expression that resolves to a **D3DX8** object.

PointsFVF
Buffer containing the vertex data around which to calculate the bounding box.

NumVertices
Number of vertices.

FVF
Combination of flexible vertex format flags that describes the vertex format.

MinArray
D3DVECTOR type, describing the returned lower-left corner of the bounding box. See Remarks.

MaxArray
D3DVECTOR type, describing the returned upper-right corner of the bounding box. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DVECTOR** values returned by **ComputeBoundingBox** are xmin, xmax, ymin, ymax, zmin, and zmax. Thus, the following defines the corners of the bounding box.

xmax, ymax, zmax
xmax, ymax, zmin
xmax, ymin, zmax

IDH_D3DX8.ComputeBoundingBox_graphicsd3dxvb

xmax, ymin, zmin
xmin, ymax, zmax
xmin, ymax, zmin
xmin, ymin, zmax
xmin, ymin, zmin

The depth of the bounding box in the z direction is zmax - zmin, in the y direction is ymax - ymin, and in the x direction is xmax - xmin. For example, with the following minimum and maximum vectors, min (-1, -1, -1) and max (1, 1, 1), the bounding box is defined in the following manner.

1, 1, 1
1, 1, -1
1, -1, 1
1, -1, -1
-1, 1, 1
-1, 1, -1
-1, -1, 1
-1, -1, -1

See Also

D3DX8.ComputeBoundingBoxFromMesh

D3DX8.ComputeBoundingBoxFromMesh

#Computes a bounding box from a mesh.

object.**ComputeBoundingBoxFromMesh(_**
MeshIn **As D3DXMesh**, _
MinArray **As D3DVECTOR**, _
MaxArray **As D3DVECTOR)**

Parts

object
Object expression that resolves to a **D3DX8** object.

MeshIn
D3DXMesh object, representing the input mesh.

MinArray
D3DVECTOR type, describing the returned lower-left corner of the bounding box. See Remarks.

MaxArray

IDH_D3DX8.ComputeBoundingBoxFromMesh_graphicsd3dxvb

D3DVECTOR type, describing the returned upper-right corner of the bounding box. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DVECTOR** values returned by **ComputeBoundingBoxFromMesh** are xmin, xmax, ymin, ymax, zmin, and zmax. Thus, the following defines the corners of the bounding box.

```
xmax, ymax, zmax
xmax, ymax, zmin
xmax, ymin, zmax
xmax, ymin, zmin
xmin, ymax, zmax
xmin, ymax, zmin
xmin, ymin, zmax
xmin, ymin, zmin
```

The depth of the bounding box in the z direction is zmax - zmin, in the y direction is ymax - ymin, and in the x direction is xmax - xmin. For example, with the following minimum and maximum vectors, min (-1, -1, -1) and max (1, 1, 1), the bounding box is defined in the following manner.

```
 1,  1,  1
 1,  1, -1
 1, -1,  1
 1, -1, -1
-1,  1,  1
-1,  1, -1
-1, -1,  1
-1, -1, -1
```

See Also

D3DX8.ComputeBoundingBox

D3DX8.ComputeBoundingSphere

#Computes a bounding sphere.

object.**ComputeBoundingSphere(** _
PointsFVF **As Any**, _
NumVertices **As Long**, _
FVF **As Long**, _
Center **As D3DVECTOR**, _
RadiusArray **As Single)**

Parts

object
Object expression that resolves to a **D3DX8** object.

PointsFVF
Buffer containing the vertex data around which to calculate the bounding sphere.

NumVertices
Number of vertices.

FVF
Combination of flexible vertex format flags that describes the vertex format.

Center
D3DVECTOR type, defining the coordinate center of the returned bounding sphere.

RadiusArray
Radius of the returned bounding sphere.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.ComputeBoundingSphereFromMesh

#Computes a bounding sphere from a mesh.

object.**ComputeBoundingSphereFromMesh(** _
MeshIn **As D3DXMesh**, _
Center **As D3DVECTOR**, _

IDH_D3DX8.ComputeBoundingSphere_graphicsd3dxvb
IDH_D3DX8.ComputeBoundingSphereFromMesh_graphicsd3dxvb

RadiusArray **As Single)**

Parts

object
Object expression that resolves to a **D3DX8** object.

MeshIn
D3DXMesh object, representing the input mesh.

Center
D3DVECTOR type, defining the coordinate center of the returned bounding sphere.

RadiusArray
Radius of the returned bounding sphere.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.ComputeNormals

#Computes normals for each vertex in a mesh.

object.**ComputeNormals(_**
Mesh **As D3DXBaseMesh)**

Parts

object
Object expression that resolves to a **D3DX8** object.

Mesh
D3DXBaseMesh object representing the normalized mesh object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Note

IDH_D3DX8.ComputeNormals_graphicsd3dxvb

The input mesh must have the D3DFVF_NORMAL flag specified in its FVF. A normal for a vertex is generated by averaging the normals of all faces that share that vertex.

D3DX8.CreateBox

#Uses a left-handed coordinate system to create a mesh containing an axis-aligned box.

object.**CreateBox(** _
D3DDevice **As Direct3DDevice8**, _
Width **As Single**, _
Height **As Single**, _
Depth **As Single**, _
RetAdjacency **As D3DXBuffer)As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

D3DDevice
A **Direct3DDevice8** object, representing the device associated with the created box mesh.

Width
Width of the box, along the x-axis.

Height
Height of the box, along the y-axis.

Depth
Depth of the box, along the z-axis.

RetAdjacency
D3DXBuffer object. When the method returns, this parameter is filled with an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

Return Values

.**D3DXMesh** object, representing the output box**.**

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

IDH_D3DX8.CreateBox_graphicsd3dxvb

D3DXERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.CreateBuffer

#Creates a buffer object.

object.**CreateBuffer(_**
NumBytes **As Long) As D3DXBuffer**

Parts

object
Object expression that resolves to a **D3DX8** object.

NumBytes
Size of the buffer to create, in bytes.

Return Values

D3DXBuffer object representing the created buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.CreateCubeTexture

#Creates an empty cube texture, adjusting the calling parameters as necessary.

object.**CreateCubeTexture(_**
Device **As Direct3DDevice8**, **_**
Size **As Long**, **_**
MipLevels **As Long**, **_**
Usage **As Long**, **_**
Format **As CONST_D3DFORMAT**, **_**
Pool As **CONST_D3DPOOL) As Direct3DCubeTexture8**

IDH_D3DX8.CreateBuffer_graphicsd3dxvb
IDH_D3DX8.CreateCubeTexture_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the cube texture.

Size
Width and height of the cube texture, in pixels. So, if the cube texture is an 8 pixel by 8 pixel cube, the value for this parameter should be 8. Note that this value must be non-zero.

MipLevels
Number of mip levels requested. If this value is zero or D3DX_DEFAULT as defined by the **CONST_D3DXENUM** enumeration, a complete mipmap chain is created.

Usage
0 or the D3DUSAGE_RENDERTARGET member of the **CONST_D3DUSAGEFLAGS** enumeration. Specifying D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **Direct3D8.CheckDeviceFormat**.

Format
Member of the **CONST_D3DFORMAT** enumeration, describing the requested pixel format for the cube texture. The returned cube texture might have a format different from that specified by *Format*. Applications should check the format of the returned cube texture.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the cube texture should be placed.

Return Values

Direct3DCubeTexture8 object representing the created cube texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DXERR_INVALIDDATA

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **Direct3DCubeTexture8.GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

Internally, **D3DX8.CreateCubeTexture** uses **D3DX8.CheckCubeTextureRequirements** to adjust the calling parameters. Therefore, calls to **D3DX8.CreateCubeTexture** often succeed where calls to **Direct3Ddevice8.CreateCubeTexture** fail.

D3DX8.CreateCubeTextureFromFile

#Creates a cube texture from a file.

object.**CreateCubeTextureFromFile(** _
Device **As Direct3DDevice8**, _
SrcFile **As String) As Direct3DCubeTexture8**

Parameters

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the cube texture.

SrcFile
String that specifies the file from which to create the cube texture. See Remarks.

Return Values

Direct3DCubeTexture8 object representing the created cube texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

IDH_D3DX8.CreateCubeTextureFromFile_graphicsd3dxvb

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Note that a resource created with this function will be placed in the memory class denoted by the D3DPOOL_MANAGED member of the **CONST_D3DPOOL** enumerated type.

CreateCubeTextureFromFile uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format.

D3DX8.CreateCubeTextureFromFileEx

#Creates a cube texture from a file. This is a more advanced method than **D3DX8.CreateCubeTextureFromFile**.

object.**CreateCubeTextureFromFileEx(_**
Device **As Direct3DDevice8**, _
SrcFile **As String**, _
TextureSize **As Long**, _
MipLevels **As Long**, _
Usage **As Long**, _
Format **As CONST_D3DFORMAT**, _
Pool As **CONST_D3DPOOL**, _
Filter **As Long**, _
MipFilter **As Long**, _
ColorKey **As Long**, _
SrcInfo **As Any**, _
Palette **As Any) As Direct3DCubeTexture8**

Parameters

object
Object expression that resolves to a **D3DX8** object.

Device

IDH_D3DX8.CreateCubeTextureFromFileEx_graphicsd3dxvb

Direct3DDevice8 object representing the device to be associated with the cube texture.

SrcFile

String that specifies the file from which to create the cube texture. See Remarks

TextureSize

Width and height of the cube texture, in pixels. So, if the cube texture is an 8 pixel by 8 pixel cube, the value for this parameter should be 8. Note that this value must be non-zero. If this value is 0 or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels

Number of mip levels requested. If this value is zero or D3DX_DEFAULT as defined by the **CONST_D3DXENUM** enumeration, a complete mipmap chain is created.

Usage

0 or the D3DUSAGE_RENDERTARGET member of the **CONST_D3DUSAGEFLAGS** enumeration. Specifying D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **Direct3D8.CheckDeviceFormat**.

Format

Member of the **CONST_D3DFORMAT** enumeration, describing the requested pixel format for the cube texture. The returned cube texture might have a format different from that specified by *Format*. Applications should check the format of the returned cube texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool

Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the cube texture should be placed.

Filter

A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

MipFilter

A combination of one or more mip filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

ColorKey

Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

SrcInfo
A **D3DXIMAGE_INFO** structure to be filled in with a description of the data in the source image file, or ByVal 0.

Palette
Object representing a 256-color palette to fill in, or Nothing.

Return Values

Direct3DCubeTexture8 object representing the created cube texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **Direct3DCubeTexture8.GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

CreateCubeTextureFromFileEx uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format.

D3DX8.CreateCubeTextureFromFileInMemory

#Creates a cube texture from a file in memory.

object.**CreateCubeTextureFromFileInMemory(** _
Device **As Direct3DDevice8**, _
SrcData **As Any**, _
LengthInBytes **As Long) As Direct3DCubeTexture8**

IDH_D3DX8.CreateCubeTextureFromFileInMemory_graphicsd3dxvb

Parameters

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the cube texture.

SrcData
File in memory from which to create the cube texture. See Remarks.

LengthInBytes
Size of the file in memory, in bytes.

Return Values

Direct3DCubeTexture8 object representing the created cube texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Note that a resource created with this function will be placed in the memory class denoted by the D3DPOOL_MANAGED member of the **CONST_D3DPOOL** enumerated type.

This method is designed to be used for loading image files stored as RT_RCDATA, which is an application-defined resource (raw data). Otherwise this method fails.

CreateCubeTextureFromFileInMemory uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format.

D3DX8.CreateCubeTextureFromFileInMemoryEx

#Creates a cube texture from a file in memory. This is a more advanced method than **D3DX8.CreateCubeTextureFromFileInMemory**.

object.**CreateCubeTextureFromFileInMemoryEx(** _
Device **As Direct3DDevice8**, _
SrcData **As Any**, _
LengthInBytes **As Long**, _
TextureSize **As Long**, _
MipLevels **As Long**, _
Usage **As Long**, _
Format **As CONST_D3DFORMAT**, _
Pool As **CONST_D3DPOOL**, _
Filter **As Long**, _
MipFilter **As Long**, _
ColorKey **As Long**, _
SrcInfo **As Any**, _
Palette **As Any) As Direct3DCubeTexture8**

Parameters

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the cube texture.

SrcData
File in memory from which to create the cube texture. See Remarks.

LengthInBytes
Size of the file in memory, in bytes.

TextureSize
Size of the file, in bytes.

MipLevels
Number of mip levels requested. If this value is zero or D3DX_DEFAULT as defined by the **CONST_D3DXENUM** enumeration, a complete mipmap chain is created.

Usage
0 or the D3DUSAGE_RENDERTARGET member of the **CONST_D3DUSAGEFLAGS** enumeration. Specifying D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of

IDH_D3DX8.CreateCubeTextureFromFileInMemoryEx_graphicsd3dxvb

the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **Direct3D8.CheckDeviceFormat**.

Format
Member of the **CONST_D3DFORMAT** enumeration, describing the requested pixel format for the cube texture. The returned cube texture might have a format different from that specified by *Format*. Applications should check the format of the returned cube texture. If this value is D3DFMT_UNKNOWN, the format is taken from the file.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the cube texture should be placed.

Filter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

MipFilter
A combination of one or more mip filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

ColorKey
Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

SrcInfo
A **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or ByVal 0.

Palette
Object representing a 256-color palette to fill in, or Nothing.

Return Values

Direct3DCubeTexture8 object representing the created cube texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **Direct3DCubeTexture8.GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**.

This method is designed to be used for loading image files stored as RT_RCDATA, which is an application-defined resource (raw data). Otherwise this method fails.

CreateCubeTextureFromFileInMemoryEx uses the DirectDrawSurface (DDS) file format. The DXTex Tool enables you to generate a cube map from other file formats and save it in the DDS file format.

D3DX8.CreateCylinder

#Uses a left-handed coordinate system to create a mesh containing a cylinder.

object.**CreateCylinder(** _
Device **As Direct3DDevice8**, _
Radius1 **As Single**, _
Radius2 **As Single**
Length **As Single**, _
Slices **As Long**, _
Stacks **As Long**, _
RetAdjacency **As D3DXBuffer) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object, representing the device associated with the created cylinder mesh.

Radius1
Radius at the negative Z end. Value should be greater than or equal to 0.0f.

Radius2

IDH_D3DX8.CreateCylinder_graphicsd3dxvb

Radius at the positive Z end. Value should be greater than or equal to 0.0f.

Length
Length of the cylinder along the z-axis.

Slices
Number of slices about the main axis.

Stacks
Number of stacks along the main axis.

RetAdjacency
D3DXBuffer object. When the method returns, this parameter is filled with an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

Return Values

D3DXMesh object, representing the output box**.**

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Cube textures differ from other surfaces in that they are collections of surfaces. To call **SetRenderTarget** with a cube texture, you must select an individual face using **Direct3DCubeTexture8.GetCubeMapSurface** and pass the resulting surface to **SetRenderTarget**

D3DX8.CreateFont

#Creates a font object for a device and font.

object.**CreateFont(** _
Device **As Direct3DDevice8**, _
hFont **As Long) As D3DXFont**

IDH_D3DX8.CreateFont_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the font object.

hFont
Handle to the font object.

Return Values

D3DXFont object representing the created font object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.CreateMesh

#Creates a mesh object using a declarator.

object.**CreateMesh(** _
NumFaces **As Long**, _
NumVertices **As Long**, _
Options **As Long**, _
Declaration **As Any**, _
Device **As Direct3DDevice8) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

NumFaces
Number of faces for the mesh. This parameter must be greater than 0.

NumVertices
Number of vertices for the mesh. This parameter must be greater than 0.

IDH_D3DX8.CreateMesh_graphicsd3dxvb

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying options for the mesh.

Declaration
A declarator that describes the vertex format for the returned mesh. This parameter must map directly to an FVF.

Device
Direct3DDevice8 object, the device to be associated with the mesh.

Return Values

D3DXMesh object representing the created mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DX8.DeclaratorFromFVF

D3DX8.CreateMeshFVF

#Creates a mesh object using a flexible vertex format (FVF) code.

object.**CreateMeshFVF(** _
NumFaces **As Long**, _
NumVertices **As Long**, _
Options **As Long**, _
FVF **As Long**, _
Device **As Direct3DDevice8) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

NumFaces

IDH_D3DX8.CreateMeshFVF_graphicsd3dxvb

Number of faces for the mesh. This parameter must be greater than 0.

NumVertices
Number of vertices for the mesh. This parameter must be greater than 0.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying options for the mesh.

FVF
Combination of flexible vertex format flags that describes the vertex format for the returned mesh.

Device
Direct3DDevice8 object, the device to be associated with the mesh.

Return Values

D3DXMesh object representing the created mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DX8.FVFFromDeclarator

D3DX8.CreatePolygon

#Uses a left-handed coordinate system to create a mesh containing an n-sided polygon.

object.**CreatePolygon(** _
D3DDevice **As Direct3DDevice8**, _
Length **As Single**, _
Sides **As Long**, _
RetAdjacency **As D3DXBuffer) As D3DXMesh**

IDH_D3DX8.CreatePolygon_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DX8** object.

D3DDevice
Direct3DDevice8 object, representing the device associated with the created polygon mesh.

Length
Length of each side

Sides
Number of sides for the polygon. Value must be greater than or equal to 3.

RetAdjacency
D3DXBuffer object. When the method returns, this parameter is filled with an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

Return Values

D3DXMesh object, representing the created polygon mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The created polygon is centered at the origin.

D3DX8.CreateRenderToSurface

#Creates a render surface.

object.**CreateRenderToSurface(** _
D3DDevice **As Direct3DDevice8**, _
Width **As Long**, _
Height **As Long**, _

IDH_D3DX8.CreateRenderToSurface_graphicsd3dxvb

Format **As CONST_D3DFORMAT**, _
DepthStencil **As Long**, _
DepthStencilFormat **As CONST_D3DFORMAT) As D3DXRenderToSurface**

Parts

object
Object expression that resolves to a **D3DX8** object.

D3DDevice
Direct3DDevice8 object, to be associated with the render surface.

Width
Width of the render surface, in pixels

Height
Height of the render surface, in pixels

Format
Member of the **CONST_D3DFORMAT** enumeration, describing the format of the render target.

DepthStencil
If True, the render surface supports a depth-stencil surface. Otherwise this member is set to False.

DepthStencilFormat
If DepthStencil is set to True, this parameter is a member of the CONST_D3DFORMAT enumeration describing the depth-stencil format of the render surface.

Return Values

D3DXRenderToSurface object, representing the created polygon mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The created polygon is centered at the origin.

D3DX8.CreateSkinMesh

#Creates an empty skin mesh object using a declarator.

object.**CreateSkinMesh(** _
NumFaces **As Long**, _
NumVertices **As Long**, _
NumBones **As Long**, _
Options **As Long**, _
Declaration **As Any**, _
D3DDevice **As Direct3DDevice8) As D3DXSkinMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

NumFaces
Number of faces for the skin mesh.

NumVertices
Number of vertices for the skin mesh.

NumBones
Number of bones for the skin mesh.

Options
A combination of one or more flags defined by the CONST_D3DXMESH enumeration, specifying options for the skin mesh.

Declaration
A **Long** value, representing the declarator that describes the vertex format for the returned skin mesh.

D3DDevice
A **Direct3DDevice8** object, the device object to be associated with the skin mesh.

Return Values

D3DXSkinMesh object, representing the created skin mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DX8.CreateSkinMesh_graphicsd3dxvb

Remarks

Use **D3DXSkinMesh.SetBoneInfluence** to populate the empty skin mesh object returned by this method.

D3DX8.CreateSkinMeshFromMesh

#Creates a skin mesh from another mesh.

object.**CreateSkinMeshFromMesh(** _
Mesh **As D3DXMesh**, _
NumBones **As Long) As D3DXSkinMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

Mesh
A **D3DXMesh** object, the mesh from which to create the skin mesh.

NumBones
Number of bones for the skin mesh.

Return Values

D3DXSkinMesh object, representing the created skin mesh object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.CreateSkinMeshFVF

#Creates an empty skin mesh object using a flexible vertex format (FVF) code.

object.**CreateSkinMeshFVF(** _
NumFaces **As Long**, _
NumVertices **As Long**, _
NumBones **As Long**, _
Options **As Long**, _
FVF **As Long**, _

IDH_D3DX8.CreateSkinMeshFromMesh_graphicsd3dxvb
IDH_D3DX8.CreateSkinMeshFVF_graphicsd3dxvb

D3DDevice **As Direct3DDevice8) As D3DXSkinMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

NumFaces
Number of faces for the skin mesh.

NumVertices
Number of vertices for the skin mesh.

NumBones
Number of bones for the skin mesh.

Options
A combination of one or more members of the **CONST_D3DXMESH** enumerated type, specifying options for the skin mesh.

FVF
Combination of flexible vertex format flags that describes the vertex format for the returned skin mesh.

D3DDevice
A **Direct3DDevice8** object, the device object to be associated with the skin mesh.

Return Values

D3DXSkinMesh object, representing the created skin mesh object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Use **D3DXSkinMesh.SetBoneInfluence** to populate the empty skin mesh object returned by this method.

D3DX8.CreateSphere

[#]Uses a left-handed coordinate system to create a mesh containing a sphere.

object.**CreateSphere(** _
D3DDevice **As Direct3DDevice8**, _
Radius **As Single**, _
Slices **As Long**, _
Stacks **As Long**, _
RetAdjacency **As D3DXBuffer) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

D3DDevice
Direct3DDevice8 object, associated with the created sphere mesh.

Radius
Radius of the sphere. This value should be greater than or equal to 0.

Slices
Number of slices about the main axis.

Stacks
Number of stacks along the main axis.

RetAdjacency
D3DXBuffer object to be filled with an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

Return Values

D3DXMesh object, representing the created sphere.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

[#] IDH_D3DX8.CreateSphere_graphicsd3dxvb

Remarks

The created sphere is centered at the origin, and its axis is aligned with the z-axis.

D3DX8.CreateSPMesh

#Creates a simplification mesh.

object.**CreateSPMesh(** _
Mesh **As D3DXMesh**, _
Adjacency **As Any**, _
VertexAttributeWeights **As Any**, _
VertexWeights **As Any) As D3DXSPMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

Mesh
D3DXMesh object, representing the mesh to simplify.

Adjacency
First element of an array of three **Long** values per face that specify the three neighbors for each face in the created simplification mesh.

VertexAttributeWeights
A **D3DXATTRIBUTEWEIGHTS** type, containing the weight for each vertex component. If this parameter is set to ByVal 0, a default type is used.

VertexWeights
First element of an array of vertex weights. Note that the higher the vertex weight for a given vertex, the less likely it is to be simplified away.

Return Values

D3DXSPMesh object, representing the created simplification mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DXERR_CANNOTATTRSORT

D3DERR_INVALIDCALL

E_OUTOFMEMORY

IDH_D3DX8.CreateSPMesh_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

A simplification mesh is used to simplify a mesh to a lower number of triangles and faces.

If *VertexAttributeWeights* is set to ByVal 0, the values are assigned according to the default **D3DXATTRIBUTEWEIGHTS** type.

```
Dim AttribruteWeights As D3DXATTRIBUTEWEIGHTS
With _
.position = 1
.Boundary = 1
.Normal = 1
.diffuse = 0
.specular = 0
.Tex(7) = 0
End With
```

This default type is what most applications should use because it considers only geometric and normal adjustment. Only in special cases will the other member fields need to be modified.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim Mesh As D3DXMesh
Dim D3DXbAdjacency As D3DXBuffer
Dim VAttWeights As Any
Dim VWeights As Any

' This code fragment assumes that all arguments
' have been properly initialized.
Call D3DX8.CreateSPMesh(Mesh, ByVal D3DXbAdjacency.GetBufferPointer, VAttWeights,
Vweights )
```

D3DX8.CreateSprite

#Creates a sprite.

object.**CreateSprite(_**
D3DDevice **As Direct3DDevice8) As D3DXSprite**

IDH_D3DX8.CreateSprite_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DX8** object.

D3DDevice
Direct3DDevice8 object representing the device to be associated with the sprite.

Return Values

D3DXSprite object representing the created sprite.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.CreateTeapot

#Creates a teapot.

object.**CreateTeapot(_**
D3DDevice **As Direct3DDevice8**, _
RetAdjacency **As D3DXBuffer) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

D3DDevice
Direct3DDevice8 object representing the device to be associated with the texture.

RetAdjacency
D3DXBuffer object. When the method returns, this parameter is filled with an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

IDH_D3DX8.CreateTeapot_graphicsd3dxvb

Return Values

D3DXMesh object representing the created teapot.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.CreateText

#Creates a mesh containing the specified text using the font associated with the device context.

object.**CreateText(** _
D3DDevice **As Direct3DDevice8**, _
HDC **As Long**, _
Text **As String**, _
Deviation **As Single**, _
Extrusion **As Single**, _
RetMesh **As D3DXMesh**, _
GlyphMetrics **As Any)**

Parts

object
Object expression that resolves to a **D3DX8** object.

D3DDevice
Direct3DDevice8 object to be associated with the created text.

HDC
Device context, containing the font for output. The font selected by the device context must be a TrueType font.

Text
A string, specifying the text to generate.

Deviation
Maximum chordal deviation from true font outlines.

Extrusion

IDH_D3DX8.CreateText_graphicsd3dxvb

Amount to extrude text in the negative Z direction.

RetMesh

The output shape, a **D3DXMesh** object.

GlyphMetrics

First element of an array of **GLYPHMETRICSFLOAT** types to receive the glyph metric data. Each **GLYPHMETRICSFLOAT** type in the array contains information about the placement and orientation of the corresponding glyph in the string. The number of elements in the array should be equal to the number of characters in the string. Note that the origin in each type is not relative to the entire string, but rather is relative to that character cell. To compute the entire bounding box, add the increment for each glyph while traversing the string.

If you are not concerned with the glyph sizes, you can pass ByVal 0 to *GlyphMetrics*

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.CreateTexture

#Creates an empty texture, adjusting the calling parameters as needed.

object.**CreateTexture(** _
Device **As Direct3DDevice8**, _
Width **As Long**, _
Height **As Long**, _
MipLevels **As Long**, _
Usage **As Long**, _
PixelFormat **As CONST_D3DFORMAT**, _
Pool **As CONST_D3DPOOL) As Direct3DTexture8**

Parts

object

Object expression that resolves to a **D3DX8** object.

IDH_D3DX8.CreateTexture_graphicsd3dxvb

Device
Direct3DDevice8 object representing the device to be associated with the texture.

Width
Width of the top-level of the texture, in pixels. The pixel dimensions of subsequent levels are the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0 (zero), 1 is taken instead. If this value is set to 0, then a value of 1 is used. D3DX_DEFAULT can also be used, see Remarks.

Height
Height of the top-level of the texture, in pixels. The pixel dimensions of subsequent levels are the truncated value of half of the previous level's pixel dimension (independently). Each dimension clamps at a size of 1 pixel. Thus, if the division by 2 results in 0 (zero), 1 is taken instead. If this value is set to 0, then a value of 1 is used. D3DX_DEFAULT can also be used, see Remarks.

MipLevels
The number of levels in the texture. If this is zero, Microsoft® Direct3D® generates all texture sub-levels down to 1×1 pixels.

Usage
0 or the D3DUSAGE_RENDERTARGET member of the **CONST_D3DUSAGEFLAGS** enumeration. Specifying D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **Direct3D8.CheckDeviceFormat**.

PixelFormat
Member of the **CONST_D3DFORMAT** enumeration, describing the requested pixel format for the texture. The returned texture might have a format different from that specified by *PixelFormat*. Applications should check the format of the returned texture.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the texture should be placed.

Return Values

Direct3DTexture8 object representing the created texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If the *Height* and *Width* parameters are both set to D3DX_DEFAULT, a value of 256 is used for both. If only one of these are set to D3DX_DEFAULT, then the texture will be square with the other dimensional parameter specifying both the height and width.

Internally, **D3DX8.CreateTexture** uses **D3DX8.CheckTextureRequirements** to adjust the calling parameters. Therefore, calls to **D3DX8.CreateTexture** often succeed where calls to **Direct3Ddevice8.CreateTexture** fail.

D3DX8.CreateTextureFromFile

#Creates a texture from a file.

object.**CreateTextureFromFile(** _
Device **As Direct3DDevice8**, _
SrcFile **As String) As Direct3DTexture8**

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the texture.

SrcFile
String that specifies the file from which to create the texture.

Return Values

Direct3DTexture8 object representing the created texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

IDH_D3DX8.CreateTextureFromFile_graphicsd3dxvb

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Mipmapped textures will automatically have each level filled with the loaded texture.

When loading images into mipmapped textures, some video cards are unable to go to a 1x1 image and this function will fail. If this happens, then the images need to be loaded manually.

Note that a resource created with this function will be placed in the memory class denoted by the D3DPOOL_MANAGED member of the **CONST_D3DPOOL** enumerated type.

D3DX8.CreateTextureFromFileEx

#Creates a texture from a file. This is a more advanced method than **D3DX8.CreateTextureFromFile**.

object.**CreateTextureFromFileEx(_**
Device **As Direct3DDevice8, _**
SrcFile **As String, _**
Width **As Long, _**
Height **As Long, _**
MipLevels **As Long, _**
Usage **As Long, _**
Format **As CONST_D3DFORMAT, _**
Pool As **CONST_D3DPOOL, _**
Filter **As Long, _**
MipFilter **As Long, _**
ColorKey **As Long, _**
SrcInfo **As Any _**
Palette **As Any) As Direct3DTexture8**

Parts

object

IDH_D3DX8.CreateTextureFromFileEx_graphicsd3dxvb

Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the texture.

SrcFile
String that specifies the file from which to create the texture.

Width
Width in pixels. If this value is zero or D3DX_DEFAULT as defined by the **D3DXENUM** enumeration, the dimensions are taken from the file.

Height
Height, in pixels. If this value is zero or D3DX_DEFAULT as defined by the **CONST_D3DXENUM** enumeration, the dimensions are taken from the file.

MipLevels
Number of mip levels requested. If this value is zero or D3DX_DEFAULT as defined by the **CONST_D3DXENUM** enumeration, a complete mipmap chain is created.

Usage
0 or the D3DUSAGE_RENDERTARGET member of the **CONST_D3DUSAGEFLAGS** enumeration. Specifying D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **Direct3D8.CheckDeviceFormat**.

Format
Member of the **CONST_D3DFORMAT** enumeration, describing the requested pixel format for the texture. The returned texture might have a format different from that specified by *Format*. Applications should check the format of the returned texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the texture should be placed.

Filter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

MipFilter
A combination of one or more mip filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

ColorKey

Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

SrcInfo

A **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or ByVal 0.

Palette

The first element of an array of 256 **PALETTEENTRY** types to fill in, or ByVal 0. See Remarks.

Return Values

Direct3DTexture8 object representing the created texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Mipmapped textures will automatically have each level filled with the loaded texture.

When loading images into mipmapped textures, some video cards are unable to go to a 1x1 image and this function will fail. If this happens, then the images need to be loaded manually.

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DX8.CreateTextureFromFileInMemory

#Creates a texture from a file in memory.

object.**CreateTextureFromFileInMemory(_**
Device **As Direct3DDevice8**, _
SrcData **As Any**, _
LengthInBytes **As Long) As Direct3DTexture8**

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the texture.

SrcData
File in memory from which to create the texture.

LengthInBytes
Size of the file in memory, in bytes.

Return Values

Direct3DTexture8 object representing the created texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DX8.CreateTextureFromFileInMemory_graphicsd3dxvb

Remarks

Note that a resource created with this function will be placed in the memory class denoted by the D3DPOOL_MANAGED member of the **CONST_D3DPOOL** enumerated type.

This method is designed to be used for loading image files stored as RT_RCDATA, which is an application-defined resource (raw data). Otherwise this method fails.

D3DX8.CreateTextureFromFileInMemoryEx

#Creates a texture from a file in memory. This is a more advanced method than **D3DX8.CreateTextureFromFileInMemory**.

object.**CreateTextureFromFileInMemoryEx(** _
Device **As Direct3DDevice8**, _
SrcData **As Any**, _
LengthInBytes **As Long**, _
Width **As Long**, _
Height **As Long**, _
MipLevels **As Long**, _
Usage **As Long**, _
Format **As CONST_D3DFORMAT**, _
Pool As **CONST_D3DPOOL**, _
Filter **As Long**, _
MipFilter **As Long**, _
ColorKey **As Long**, _
SrcInfo **As Any**, _
Palette **As Any) As Direct3DTexture8**

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the texture.

SrcData
File in memory from which to create the texture.

LengthInBytes
Size of the file in memory, in bytes.

Width

IDH_D3DX8.CreateTextureFromFileInMemoryEx_graphicsd3dxvb

Width in pixels. If this value is zero or D3DX_DEFAULT as defined by the **CONST_D3DXENUM** enumeration, the dimensions are taken from the file.

Height

Height, in pixels. If this value is zero or D3DX_DEFAULT as defined by the **CONST_D3DXENUM** enumeration, the dimensions are taken from the file.

MipLevels

Number of mip levels requested. If this value is zero or D3DX_DEFAULT as defined by the **CONST_D3DXENUM** enumeration, a complete mipmap chain is created.

Usage

0 or the D3DUSAGE_RENDERTARGET member of the **CONST_D3DUSAGEFLAGS** enumeration. Specifying D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **Direct3D8.CheckDeviceFormat**.

Format

Member of the **CONST_D3DFORMAT** enumeration, describing the requested pixel format for the texture. The returned texture might have a format different from that specified by *Format*. Applications should check the format of the returned texture. If *Format* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool

Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the texture should be placed.

Filter

A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

MipFilter

A combination of one or more mip filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

ColorKey

Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

SrcInfo

A **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or ByVal 0.

Palette

The first element of an array of 256 **PALETTEENTRY** types to fill in, or ByVal 0. See Remarks.

Return Values

Direct3DTexture8 object representing the created texture.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

This method is designed to be used for loading image files stored as RT_RCDATA, which is an application-defined resource (raw data). Otherwise this method fails.

D3DX8.CreateTextureFromResource

#Creates a texture from a resource.

object.**CreateTextureFromResource(_**
Device **As Direct3DDevice8**, _
hModule **As Long**, _
SrcResource **As String) As Direct3DTexture8**

Parts

object

IDH_D3DX8.CreateTextureFromResource_graphicsd3dxvb

Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the texture.

hModule
Handle to the module where the resource is located, or App.hInstance if the resource information is located in the current process.

SrcResource
String that specifies the resource from which to create the texture.

Return Values

Direct3DTexture8 object representing the created texture object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Note that a resource created with this function will be placed in the memory class denoted by the D3DPOOL_MANAGED member of the **CONST_D3DPOOL** enumerated type.

See Also

D3DX8.CreateTextureFromResourceEx

D3DX8.CreateTextureFromResourceEx

#Creates a texture from a resource. This is a more advanced method than **D3DX8.CreateTextureFromResource**.

IDH_D3DX8.CreateTextureFromResourceEx_graphicsd3dxvb

object.**CreateTextureFromResourceEx(** _
Device **As Direct3DDevice8**, _
hSrcModule **As Long**, _
SrcResource **As String**, _
Width **As Long**, _
Height **As Long**, _
MipLevels **As Long**, _
Usage **As Long**, _
PixelFormat **As CONST_D3DFORMAT**, _
Pool As **CONST_D3DPOOL**, _
Filter **As Long**, _
MipFilter **As Long**, _
ColorKey **As Long,** _
SrcInfo **As Any,** _
Palette **As Any) As Direct3DTexture8**

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the texture.

hSrcModule
Handle to the module where the resource is located, or App.hInstance if the resource information is located in the current process.

SrcResource
String that specifies the resource from which to create the texture.

Width
Width in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

Height
Height, in pixels. If this value is zero or D3DX_DEFAULT, the dimensions are taken from the file.

MipLevels
Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage
0 or the D3DUSAGE_RENDERTARGET member of the **CONST_D3DUSAGEFLAGS** enumeration. Specifying D3DUSAGE_RENDERTARGET indicates that the surface is to be used as a render target. The resource can be passed to the *NewRenderTarget* parameter of the **SetRenderTarget** method. If D3DUSAGE_RENDERTARGET is specified, the application should check that the device supports this operation by calling **Direct3D8.CheckDeviceFormat**.

PixelFormat
Member of the **CONST_D3DFORMAT** enumeration, describing the requested pixel format for the texture. The returned texture might have a format different from that specified by *PixelFormat*. Applications should check the format of the returned texture. If *PixelFormat* is D3DFMT_UNKNOWN, the format is taken from the file.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the texture should be placed.

Filter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

MipFilter
A combination of one or more mip filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

ColorKey
Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

SrcInfo
A **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or ByVal 0.

Palette
The first element of an array of 256 **PALETTEENTRY** types to fill in, or ByVal 0. See Remarks.

Return Values

Direct3DTexture8 object representing the created texture object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

See Also

D3DX8.CreateTextureFromResource

D3DX8.CreateTorus

#Creates an empty texture.

object.**CreateTorus(** _
D3DDevice **As Direct3DDevice8**, _
InnerRadius **As Single**, _
OuterRadius **As Single**, _
Sides **As Long**, _
Rings **As Long**, _
RetAdjacency **As D3DXBuffer) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

D3DDevice
Direct3DDevice8 object representing the device to be associated with the texture.

InnerRadius
Inner-radius of the torus. Value should be greater than or equal to 0.

OuterRadius
Outer-radius of the torus. Value should be greater than or equal to 0.

Sides
Number of sides in a cross-section. Value must be greater than or equal to 3.

Rings
Number of rings making up the torus. Value must be greater than or equal to 3.

IDH_D3DX8.CreateTorus_graphicsd3dxvb

RetAdjacency
D3DXBuffer object. When the method returns, this parameter is filled with an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

Return Values

D3DXMesh object representing the created torus.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **CreateTorus** method draws a doughnut, centered at (0, 0, 0) whose axis is aligned with the z-axis. The inner-radius of the torus is the radius of the cross-section (the minor-radius), and the outer-radius of the torus is the radius of the central hole.

This method returns a mesh that can be used later for drawing or manipulation by the application.

D3DX8.CreateVolumeTexture

#Creates an empty volume texture, adjusting the calling parameters as necessary.

object.**CreateVolumeTexture(_**
Device **As Direct3DDevice8**, **_**
Width **As Long**, **_**
Height **As Long**, **_**
Depth **As Long**, **_**
MipLevels **As Long**, **_**
Usage **As Long**, **_**
PixelFormat **As CONST_D3DFORMAT**, **_**
Pool As **CONST_D3DPOOL) As Direct3DVolume8**

IDH_D3DX8.CreateVolumeTexture_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DX8** object.

Device
Direct3DDevice8 object representing the device to be associated with the volume.

Width
Width in pixels. This value must be non-zero.

Height
Height in pixels. This value must be non-zero.

Depth
Depth in pixels. This value must be non-zero.

MipLevels
Number of mip levels requested. If this value is zero or D3DX_DEFAULT, a complete mipmap chain is created.

Usage
Currently not used, set to 0.

PixelFormat
Member of the **CONST_D3DFORMAT** enumeration, describing the requested pixel format for the volume texture. The returned volume texture might have a format different from that specified by *Format*. Applications should check the format of the returned volume texture.

Pool
Member of the **CONST_D3DPOOL** enumeration, describing the memory class into which the volume texture should be placed.

Return Values

Direct3DVolume8 object representing the created volume texture object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Internally, **D3DX8.CreateVolumeTexture** uses **D3DX8.CheckVolumeTextureRequirements** to adjust the calling parameters. Therefore, calls to **D3DX8.CreateVolumeTexture** often succeed where calls to **Direct3Ddevice8.CreateVolumeTexture** fail.

D3DX8.DeclaratorFromFVF

#Returns a declarator from a flexible vertex format (FVF) code.

object.**DeclaratorFromFVF(_**
FVF **As Long**, **_**
Declarator **As D3DXDECLARATOR)**

Parts

object
Object expression that resolves to a **D3DX8** object.

FVF
Combination of flexible vertex format flags that describes the FVF from which to generate the returned declarator.

Declarator
A **D3DXDECLARATOR** type, describing the returned declarator.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DX8.FVFFromDeclarator

D3DX8.DrawText

#Draws formatted text on a Microsoft® Direct3D® device.

object.**DrawText(_**
D3dFont **As D3DXFont**, **_**
Color **As Long**, **_**
TextString **As String**, **_**

IDH_D3DX8.DeclaratorFromFVF_graphicsd3dxvb
IDH_D3DX8.DrawText_graphicsd3dxvb

RECT **As RECT**, _
Format **As Long)**

Parts

object
Object expression that resolves to a **D3DX8** object.

D3dFont
A **D3DXFont** object specifying the font to use.

Color
Long value specifying the color of the text.

TextString
The string to draw.

RECT
RECT type that contains the rectangle, in logical coordinates, in which the text is to be formatted.

Format
A combination of values from the CONST_DTFLAGS enumeration, specifying the text formatting.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft Visual Basic® Error Handling topic.

See Also

CONST_DTFLAGS, **D3DXFont.DrawTextW**

D3DX8.FilterCubeTexture

#Filters a cube texture.

object.**FilterCubeTexture(** _
CubeTexture **As Direct3DCubeTexture8**, _

IDH_D3DX8.FilterCubeTexture_graphicsd3dxvb

Palette **As Any**, _
SrcLevel **As Long**, _
MipFilter **As Long)**

Parts

object
Object expression that resolves to a **D3DX8** object.

CubeTexture
Direct3DCubeTexture8 object representing the cube texture to filter.

Palette
The first element of an array of 256 **PALETTEENTRY** types to fill in, or ByVal 0. If a palette is not specified, the default Microsoft® Direct3D® palette (an all opaque white palette) is provided. See Remarks.

SrcLevel
The level whose image is used to generate the subsequent levels. Specifying D3DX_DEFAULT for this parameter is equivalent to specifying 0.

MipFilter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the mipmap is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

For each side of the cube texture, a box-filter is recursively applied to each level to generate the next level.

D3DX8.FilterTexture

#Filters mipmap levels of a texture.

object.**FilterTexture(** _
Texture **As Direct3DTexturep8**, _
Palette **As Any**, _
SrcLevel **As Long**, _
MipFilter **As Long)**

Parts

object
Object expression that resolves to a **D3DX8** object.

Texture
Direct3DTexture8 object representing the texture to filter.

Palette
The first element of an array of 256 **PALETTEENTRY** types to fill in, or ByVal 0 for nonpalletized formats. If a palette is not specified, the default Microsoft® Direct3D® palette (an all opaque white palette) is provided. See Remarks.

SrcLevel
The level whose image is used to generate the subsequent levels. Specifying D3DX_DEFAULT for this parameter is equivalent to specifying 0.

MipFilter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the mipmap is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

IDH_D3DX8.FilterTexture_graphicsd3dxvb

Box-filter is recursively applied to each texture level to generate the next texture level.

D3DX8.FilterVolumeTexture

#Filters mipmap levels of a volume texture.

object.**FilterVolumeTexture(** _
VolumeTexture **As Direct3DVolume8**, _
Palette **As Any**, _
SrcLevel **As Long**, _
MipFilter **As Long)**

Parts

object
Object expression that resolves to a **D3DX8** object.

VolumeTexture
Direct3DVolume8 object representing the volume to filter.

Palette
The first element of an array of 256 **PALETTEENTRY** types to fill in, or ByVal 0 for nonpalletized formats. If a palette is not specified, the default Microsoft® Direct3D® palette (an all opaque white palette) is provided. See Remarks.

SrcLevel
The level whose image is used to generate the subsequent levels. Specifying D3DX_DEFAULT for this parameter is equivalent to specifying 0.

MipFilter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the mipmap is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_BOX.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the

IDH_D3DX8.FilterVolumeTexture_graphicsd3dxvb

Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

Box-filter is recursively applied to each texture level to generate the next texture level.

D3DX8.FVFFromDeclarator

#Returns a flexible vertex format (FVF) code from a declarator.

object.**FVFFromDeclarator(_**
Declarator **As D3DXDECLARATOR) As Long**

Parts

object
Object expression that resolves to a **D3DX8** object.

Declarator
A **D3DXDECLARATOR** type, describing the declarator from which to generate the FVF code.

Return Values

Long value, representing the returned combination of flexible vertex format flags that describes the vertex format returned from the declarator.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Note

This function fails for any declarator that does not map directly to an FVF.

See Also

D3DX8.DeclaratorFromFVF

D3DX8.GeneratePMesh

#Generates a progressive mesh.

object.**GeneratePMesh(_**

IDH_D3DX8.FVFFromDeclarator_graphicsd3dxvb
IDH_D3DX8.GeneratePMesh_graphicsd3dxvb

Mesh **As D3DXMesh**, _
Adjacency **As Any**, _
VertexAttributeWeights **As Any**, _
VertexWeights **As Any**, _
MinValue **As Long**, _
Options **As Long) As D3DXPMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

Mesh
D3DXMesh object, representing the source mesh.

Adjacency
First element of an array of three **Long** values per face that specify the three neighbors for each face in the created progressive mesh.

VertexAttributeWeights
a **D3DXATTRIBUTEWEIGHTS** type, containing the weight for each vertex component. If this parameter is set to ByVal 0, a default type is used. See Remarks.

VertexWeights
First element of an array of vertex weights. If this parameter is set to ByVal 0, all vertex weights are set to 1.0. Note that the higher the vertex weight for a given vertex, the less likely it is to be simplified away.

MinValue
Number of vertices or faces, depending on the which flag is set in the *Options* parameter, by which to simplify the source mesh.

Options
Specifies simplification options for the mesh. One of the following flags defined by the **CONST_D3DXMESHSIMP** enumeration can be set.

Return Values

D3DXPMesh object, representing the created progressive mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_CANNOTATTRSORT

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method generates a mesh where the level of detail (LOD) can be adjusted from the current value to the *MinValue*.

If the simplification process cannot reduce the mesh to *MinValue*, the call still succeeds, because *MinValue* is a desired minimum, not an absolute minimum.

If *VertexAttributeWeights* is set to ByVal 0, the values are assigned according to the default **D3DXATTRIBUTEWEIGHTS** type.

```
Dim AttribruteWeights As D3DXATTRIBUTEWEIGHTS
With _
.position = 1
.Boundary = 1
.Normal = 1
.diffuse = 0
.specular = 0
.Tex(7) = 0
End With
```

This default type is what most applications should use because it considers only geometric and normal adjustment. Only in special cases do the other member fields need to be modified.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim Mesh As D3DXMesh
Dim D3DXbAdjacency As D3DXBuffer
Dim VAttWeights As Any
Dim VWeights As Any
Dim MinV As Long
Dim Opts As Long

' This code fragment assumes that all arguments
' have been properly initialized.
Call D3DX8.GeneratePMesh(Mesh, ByVal D3DXbAdjacency.GetBufferPointer, VAttWeights,
Vweights, MinV, Opts )
```

D3DX8.GetErrorString

#Returns the error string for an error code.

IDH_D3DX8.GetErrorString_graphicsd3dxvb

object.**GetErrorString(_**
ErrorCode **As Long) As String**

Parts

object
Object expression that resolves to a **D3DX8** object.
ErrorCode
The specified error code to decipher.

Return Values

Returned error string.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft Visual Basic® Error Handling topic.

Remarks

This function interprets all Microsoft® Direct3D® error codes.

D3DX8.GetFVFVertexSize

#Returns the size of a vertex for a flexible vertex format.

object.**GetFVFVertexSize(_**
FVF **As Long) As Long**

Parts

object
Object expression that resolves to a **D3DX8** object.
FVF
Flexible vertex format to be queried. A combination of flexible vertex format flags.

Return Values

The flexible vertex format vertex size, in bytes.

IDH_D3DX8.GetFVFVertexSize_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.Intersect

#Determines if a ray intersects with a mesh.

object.**Intersect(** _
MeshIn **As D3DXMesh**, _
RayPos **As D3DVECTOR**, _
RayDir **As D3DVECTOR**, _
retHit **As Long**, _
retFaceIndex **As Long**, _
retU **As Single**, _
retV **As Single**, _
retDist **As Single)**

Parts

object
Object expression that resolves to a **D3DX8** object.

MeshIn
D3DXMesh object, representing the mesh to be tested.

RayPos
D3DVECTOR type, specifying the origin coordinate of the ray.

RayDir
D3DVECTOR type, specifying the direction of the ray.

retHit
If the ray intersects a triangular face on the mesh, this value is set to TRUE. Otherwise, this value is set to FALSE.

retFaceIndex
Index value of the face closest to the ray origin, if *retHit* is TRUE.

retU
Barycentric hit coordinate.

retV
Barycentric hit coordinate.

retDist
Ray intersection parameter distance.

IDH_D3DX8.Intersect_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DX8.LoadPMeshFromFile

#Loads a progressive mesh from a file.

object.**LoadPMeshFromFile(** _
Filename **As String**, _
D3DDevice **As Direct3DDevice8**, _
RetMaterials **As D3DXBuffer**, _
RetNumMaterials **As Long) As D3DXPMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

Filename
String the specifies the name of the file from which to load the progressive mesh.

D3DDevice
Direct3DDevice8 object, the device object associated with the progressive mesh.

RetMaterials
D3DXBuffer object. When this method returns, this parameter is filled with an array of **D3DXMATERIAL** types, containing information saved in the file.

RetNumMaterials
The number of **D3DXMATERIAL** types in the *RetMaterials* array, when the method returns.

Return Values

D3DXPMesh object representing the loaded progressive mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DXERR_INVALIDMESH.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DX8.LoadPMeshFromFile_graphicsd3dxvb

See Also

D3DX8.SavePMeshToFile

D3DX8.LoadSkinMeshFromXof

#Loads a mesh from a Microsoft® DirectX® (.x) file data object.

object.**LoadSkinMeshFromXof(** _
OfObjMesh **As Unknown**, _
Options **As Long**, _
D3DDevice **As Direct3DDevice8**, _
RetAdjacency **As D3DXBuffer**, _
RetMaterials **As D3DXBuffer**, _
RetMatout **As Long**, _
RetBoneNames **As D3DXBuffer**, _
RetBoneTransforms **As D3DXBuffer) As D3DXSkinMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

OfObjMesh
DirectXFileData object representing the DirectX file data object to load.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

D3DDevice
Direct3DDevice8 object, the device object associated with the mesh.

RetAdjacency
D3DXBuffer object. When the method returns, this parameter is filled with an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

RetMaterials
D3DXBuffer object. When this method returns, this parameter is filled with an array of **D3DXMATERIAL** types, containing information saved in the DirectX file.

RetMatout
The number of **D3DXMATERIAL** types in the *RetMaterials* array, when the method returns.

RetBoneNames
A **D3DXBuffer** object, containing the bone names.

RetBoneTransforms

IDH_D3DX8.LoadSkinMeshFromXof_graphicsd3dxvb

A **D3DXBuffer** object, containing the bone transforms.

Return Values

D3DXSkinMesh object representing the loaded mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft Visual Basic® Error Handling topic.

D3DX8.LoadMeshFromX

#Loads a mesh from a Microsoft® DirectX® (.x) file.

object.**LoadMeshFromX(** _
FileName **As String**, _
Options **As Long**, _
Device **As Direct3DDevice8**, _
RetAdjacency **As D3DXBuffer**, _
RetMaterials **As D3DXBuffer**, _
RetMaterialCount **As Long) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

FileName
String that specifies the name of the DirectX file to load.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

Device
Direct3DDevice8 object, the device associated with the mesh.

RetAdjacency

IDH_D3DX8.LoadMeshFromX_graphicsd3dxvb

D3DXBuffer object. When the method returns, this parameter is filled with an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

RetMaterials

D3DXBuffer object. When this method returns, this parameter is filled with an array of **D3DXMATERIAL** types, containing information saved in the DirectX file.

RetMaterialCount

Number of **D3DXMATERIAL** types in the *RetMaterials* array, when the method returns.

Return Values

D3DXMesh object representing the loaded mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The following code fragment shows how to call **LoadMeshFromX**.

```
Dim g_device As Direct3DDevice8

Dim g_d3dx As D3DX8
Set g_d3dx = New D3DX8

Dim mesh As D3DXMesh
Dim matBuf As D3DXBuffer

' The following assumes that FileName, g_device, matBuf, and matCount have been set to valid values.
Set mesh = g_d3dx.LoadMeshFromX(FileName, D3DXMESH_MANAGED, g_device, Nothing, matBuf, matCount)
```

When **LoadMeshFromX** returns, matBuf and matCount have been set to valid values.

Note

All the meshes in the file are collapsed into one output mesh. If the file contains a frame hierarchy, all the transformations are applied to the mesh.

See Also

D3DX8.BufferGetMaterial, **D3DX8.BufferGetTextureName**

D3DX8.LoadMeshFromXof

#Loads a mesh from a Microsoft® DirectX® (.x) file data object.

object.**LoadMeshFromXof(** _
ofObjMesh **As Unknown**, _
Options **As Long**, _
Device **As Direct3DDevice8**, _
RetBufAdjacency **As D3DXBuffer**, _
RetMaterials **As D3DXBuffer**, _
RetMaterialCount **As Long) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

ofObjMesh
DirectXFileData object representing the DirectX file data object to load.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

Device
Direct3DDevice8 object, the device object associated with the mesh.

RetBufAdjacency
D3DXBuffer object. When the method returns, this parameter is filled with an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

RetMaterials
D3DXBuffer object. When this method returns, this parameter is filled with an array of **D3DXMATERIAL** types, containing information saved in the DirectX file.

RetMaterialCount
Number of **D3DXMATERIAL** types in the *RetMaterials* array, when the method returns.

IDH_D3DX8.LoadMeshFromXof_graphicsd3dxvb

Return Values

D3DXMesh object representing the loaded mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft Visual Basic® Error Handling topic.

See Also

DirectXFileData

D3DX8.LoadSurfaceFromFile

#Loads a surface from a file.

object.**LoadSurfaceFromFile(** _
DestSurface **As Direct3DSurface8**, _
DestPalette **As Any**, _
DestRect **As Any**, _
SrcFile **As String**, _
SrcRect **As Any**, _
Filter **As Long**, _
ColorKey **As Long**, _
SrcInfo**)**

Parts

object
Object expression that resolves to a **D3DX8** object.

DestSurface
Direct3DSurface8 object. Specifies the destination surface, which receives the image.

DestPalette
The first element of an array of 256 **PALETTEENTRY** types, representing the destination palette, or ByVal 0. See Remarks.

IDH_D3DX8.LoadSurfaceFromFile_graphicsd3dxvb

DestRect
RECT type. Specifies the destination rectangle. Set this parameter to ByVal 0 to specify the entire surface.

SrcFile
String that specifies the file name of the source image.

SrcRect
A **RECT** type. Specifies the source rectangle. Set this parameter to ByVal 0 to specify the entire image.

Filter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

ColorKey
Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

SrcInfo
A **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or ByVal 0.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DX8.LoadSurfaceFromFileInMemory

#Loads a surface from a file in memory.

IDH_D3DX8.LoadSurfaceFromFileInMemory_graphicsd3dxvb

object.**LoadSurfaceFromFileInMemory(** _
DestSurface **As Direct3DSurface8**, _
DestPalette **As Any**, _
DestRect **As Any**, _
SrcData **As Any**, _
LengthInBytes **As Long**, _
SrcRect **As Any**, _
Filter **As Long**, _
ColorKey **As Long**, _
SrcInfo **As Any)**

Parts

object
Object expression that resolves to a **D3DX8** object.

DestSurface
Direct3DSurface8 object. Specifies the destination surface, which receives the image.

DestPalette
The first element of an array of 256 **PALETTEENTRY** types, representing the destination palette, or ByVal 0. See Remarks.

DestRect
RECT type. Specifies the destination rectangle. Set this parameter to ByVal 0 to specify the entire surface.

SrcData
File in memory from which to load the surface.

LengthInBytes
Size of the file in memory, in bytes.

SrcRect
A **RECT** type. Specifies the source rectangle. Set this parameter to ByVal 0 to specify the entire image.

Filter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

ColorKey
Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

SrcInfo
A **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or ByVal 0.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DX8.LoadSurfaceFromMemory

#Loads a surface from memory.

object.**LoadSurfaceFromMemory(_**
DestSurface **As Direct3DSurface8**, _
DestPalette **As Any**, _
DestRect **As Any**, _
SrcData **As Any**, _
SrcFormat **As CONST_D3DFORMAT**, _
SrcPitch **As Long**, _
SrcPalette **As Any**, _
SrcRect **As Any**, _
Filter **As Long**, _
ColorKey **As Long)**

Parts

object
Object expression that resolves to a **D3DX8** object.

DestSurface
Direct3DSurface8 object. Specifies the destination surface, which receives the image.

DestPalette

IDH_D3DX8.LoadSurfaceFromMemory_graphicsd3dxvb

The first element of an array of 256 **PALETTEENTRY** types, representing the destination palette, or ByVal 0. See Remarks.

DestRect
RECT type. Specifies the destination rectangle.

SrcData
The top-left corner of the source image in memory.

SrcFormat
Member of the **CONST_D3DFORMAT** enumeration; the pixel format of the source image.

SrcPitch
Pitch of source image, in bytes. For DXT formats, this number should represent the width of one row of cells, in bytes.

SrcPalette
The first element of an array of 256 **PALETTEENTRY** types, representing the source palette, or ByVal 0.

SrcRect
RECT type, describing the dimensions of the source image in memory.

Filter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

ColorKey
Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented

in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DX8.LoadSurfaceFromResource

#Loads a surface from a resource.

object.**LoadSurfaceFromResource(** _
DestSurface **As Direct3DSurface8**, _
DestPalette **As Any**, _
DestRect **As Any**, _
hSrcModule **As Long**, _
SrcResource **As String**, _
SrcRect **As Any**, _
Filter **As Long**, _
ColorKey **As Long**, _
SrcInfo **As Any)**

Parts

object
Object expression that resolves to a **D3DX8** object.

DestSurface
Direct3DSurface8 object. Specifies the destination surface, which receives the image.

DestPalette
The first element of an array of 256 **PALETTEENTRY** types, representing the destination palette, or ByVal 0. See Remarks.

DestRect
RECT type. Specifies the destination rectangle. Set this parameter to ByVal 0 to specify the entire surface.

hSrcModule
Handle to the module where the resource is located, or App.hInstance if the resource information is located in the current process.

SrcResource
String that specifies the resource name of the source image.

SrcRect
RECT type, describing the dimensions of the source image in memory.

Filter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

IDH_D3DX8.LoadSurfaceFromResource_graphicsd3dxvb

ColorKey
Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

SrcInfo
A **D3DXIMAGE_INFO** structure to be filled with a description of the data in the source image file, or ByVal 0.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DX8.LoadSurfaceFromSurface

#Loads a surface from another surface with color conversion.

object.**LoadSurfaceFromSurface(** _
DestSurface **As Direct3DSurface8**, _
DestPalette **As Any**, _
DestRect **As Any**, _
SrcSurface **As Direct3DSurface8**, _
SrcPalette **As Any**, _
SrcRect **As Any**, _
Filter **As Long**, _
ColorKey **As Long)**

Parts

object

IDH_D3DX8.LoadSurfaceFromSurface_graphicsd3dxvb

Object expression that resolves to a **D3DX8** object.

DestSurface
Direct3DSurface8 object. Specifies the destination surface, which receives the image.

DestPalette
The first element of an array of 256 **PALETTEENTRY** types, representing the destination palette, or ByVal 0. See Remarks.

DestRect
RECT type. Specifies the destination rectangle. Set this parameter to ByVal 0 to specify the entire surface.

SrcSurface
Direct3DSurface8 object, representing the source surface.

SrcPalette
The first element of an array of 256 **PALETTEENTRY** types, representing the source palette, or ByVal 0.

SrcRect
RECT type. Specifies the source rectangle. Set this parameter to ByVal 0 to specify the entire surface.

Filter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

ColorKey
Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented

in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DX8.LoadVolumeFromMemory

#Loads a volume from memory.

object.**LoadVolumeFromMemory(** _
DestVolume **As Direct3DVolume8**, _
DestPalette **As Any**, _
DestBox **As Any**, _
SrcMemory **As Any**, _
SrcFormat **As Long**, _
SrcRowPitch **As Long**, _
SrcSlicePitch **As Long**, _
SrcPalette **As Any**, _
SrcBox **As D3DBOX**, _
Filter **As Long**, _
ColorKey **As Long)**

Parts

object
Object expression that resolves to a **D3DX8** object.

DestVolume
Direct3DVolume8 object. Specifies the destination volume, which receives the image.

DestPalette
The first element of an array of 256 **PALETTEENTRY** types, representing the destination palette, or ByVal 0. See Remarks.

DestBox
D3DBOX type. Specifies the destination box. Set this parameter to ByVal 0 to specify the entire volume.

SrcMemory
Value that indicates the top-left corner of the source volume in memory.

SrcFormat
Member of the **CONST_D3DFORMAT** enumeration; the pixel format of the source volume.

SrcRowPitch
Pitch of source image, in bytes. For DXT formats (compressed texture formats), this number should represent the size of one row of cells, in bytes.

SrcSlicePitch

IDH_D3DX8.LoadVolumeFromMemory_graphicsd3dxvb

Pitch of source image, in bytes. For DXT formats (compressed texture formats), this number should represent the size of one slice of cells, in bytes.

SrcPalette

The first element of an array of 256 **PALETTEENTRY** types, representing the source palette, or ByVal 0.

SrcBox

D3DBOX type. Specifies the source box. ByVal 0 is not a valid value for this parameter.

Filter

A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

ColorKey

Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DX8.LoadVolumeFromVolume

#Loads a volume from another volume.

object.**LoadVolumeFromVolume(** _
DestVolume **As Direct3DVolume8**, _

IDH_D3DX8.LoadVolumeFromVolume_graphicsd3dxvb

DestPalette **As Any**, _
DestBox **As Any**, _
SrcVolume **As Direct3DVolume8**, _
SrcPalette **As Any**, _
SrcBox **As Any**, _
Filter **As Long**, _
ColorKey **As Long)**

Parts

object
Object expression that resolves to a **D3DX8** object.

DestVolume
Direct3DVolume8 object. Specifies the destination volume, which receives the image.

DestPalette
The first element of an array of 256 **PALETTEENTRY** types, representing the destination palette, or ByVal 0. See Remarks.

DestBox
D3DBOX type. Specifies the destination box. Set this parameter to ByVal 0 to specify the entire volume.

SrcVolume
Direct3DVolume8 object. Specifies the source volume.

SrcPalette
The first element of an array of 256 **PALETTEENTRY** types, representing the source palette, or ByVal 0.

SrcBox
D3DBOX type. Specifies the source box. Set this parameter to ByVal 0 to specify the entire volume.

Filter
A combination of one or more filter flags defined by the **CONST_D3DXENUM** enumeration, controlling how the image is filtered. Specifying D3DX_DEFAULT for this parameter is the equivalent of specifying D3DX_FILTER_DITHER **Or** D3DX_FILTER_TRIANGLE.

ColorKey
Value to replace with transparent black, or 0 to disable the colorkey. This is always a 32-bit ARGB color, independent of the source image format. Alpha is significant and should usually be set to FF for opaque colorkeys. Thus, for opaque black, the value would be equal to &HFF000000.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

For more information on **PALETTEENTRY** see the Microsoft Platform Software Development Kit (SDK). Note that as of Microsoft DirectX® 8.0, the *peFlags* member of the **PALETTEENTRY** structure does not work the way it is documented in the Platform SDK. The *peFlags* member is now the alpha channel for 8-bit palletized formats.

D3DX8.SaveMeshToX

#Saves a mesh to a Microsoft® DirectX® (.x) file.

object.**SaveMeshToX(** _
FileName **As String**, _
Mesh **As D3DXMesh**, _
AdjacencyArray **As Any**, _
MaterialArray **As D3DXMATERIAL**, _
MaterialCount **As Long**, _
xFormat **As Long)**

Parts

object
Object expression that resolves to a **D3DX8** object.

FileName
String that specifies the name of the DirectX file identifying the saved mesh.

Mesh
D3DXMesh object representing the mesh to save to a DirectX file.

AdjacencyArray
First element of an array of three **Long** values per face that specify the three neighbors for each face in the mesh.

MaterialArray
First element of an array of **D3DXMATERIAL** types, containing material information to be saved in the DirectX file.

MaterialCount
Number of **D3DXMATERIAL** types in the *MaterialArray array*.

xFormat

IDH_D3DX8.SaveMeshToX_graphicsd3dxvb

A member of the **CONST_DXFILEFORMATFLAGS** enumeration indicating the format to use when saving the DirectX file.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The default value for the file format is DXFILEFORMAT_BINARY. The file format values can be combined together in a logical **OR** to create compressed text or compressed binary files. If a file is specified as both binary (0) and text (1), it is saved as a text file because the value is indistinguishable from the text file format value (0 + 1 = 1). If you indicate that the file format should be text and compressed, the file is first written out as text and then compressed. However, compressed text files are not as efficient as binary text files, so in most cases you indicate binary and compressed. Setting a file to be compressed without specifying a format results in a binary, compressed file.

D3DX8.SavePMeshToFile

#Saves a progressive mesh to a file.

object.**SavePMeshToFile(** _
Filename **As String**, _
Mesh **As D3DXPMesh**, _
MaterialArray **As D3DXMATERIAL**, _
MaterialCount **As Long)**

Parts

object
Object expression that resolves to a **D3DX8** object.

Filename
String that specifies the name of the file to which to save the progressive mesh.

Mesh
D3DXPMesh object, representing the progressive mesh to be saved to a file.

MaterialArray
First element of an array of **D3DXMATERIAL** types, containing material information to be saved in the file.

MaterialCount

IDH_D3DX8.SavePMeshToFile_graphicsd3dxvb

Number of **D3DXMATERIAL** types in *MaterialArray*.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DXERR_INVALIDMESH.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DX8.LoadPMeshFromFile

D3DX8.SimplifyMesh

#Simplifies a mesh.

object.**SimplifyMesh(** _
Mesh **As D3DXMesh**, _
Adjacency **As Any**, _
VertexAttributeWeights **As Any**, _
VertexWeights **As Any**, _
MinValue **As Long**, _
Options **As Long) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

Mesh
D3DXMesh object, representing the source mesh.

Adjacency
First element of an array of three **Long** values per face that specify the three neighbors for each face in the mesh to be simplified.

VertexAttributeWeights
A **D3DXATTRIBUTEWEIGHTS** type, containing the weight for each vertex component. If this parameter is set to ByVal 0, a default type is used. See Remarks.

VertexWeights
First element of an array of vertex weights. If this parameter is set to ByVal 0, all vertex weights are set to 1.0.

MinValue

IDH_D3DX8.SimplifyMesh_graphicsd3dxvb

Number of vertices or faces, depending on the which flag is set in the *Options* parameter, by which to simplify the source mesh.

Options

Specifies simplification options for the mesh. One of the flags defined by the **CONST_D3DXMESHSIMP** enumeration can be set.

Return Values

D3DXMesh object, representing the returned simplification mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method generates a mesh that has *MinValue* vertices or faces.

If the simplification process cannot reduce the mesh to *MinValue*, the call still succeeds, because *MinValue* is a desired minimum, not an absolute minimum.

If *VertexAttributeWeights* is set to ByVal 0, the values are assigned according to the default **D3DXATTRIBUTEWEIGHTS** type.

```
Dim AttribruteWeights As D3DXATTRIBUTEWEIGHTS
With _
.position = 1
.Boundary = 1
.Normal = 1
.diffuse = 0
.specular = 0
.Tex(7) = 0
End With
```

This default type is what most applications should use because it considers only geometric and normal adjustment. Only in special cases do the other member fields need to be modified.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim Mesh As D3DXMesh
Dim D3DXbAdjacency As D3DXBuffer
Dim VAttWeights As Any
Dim VWeights As Any
Dim MinV As Long
Dim Opts As Long

' This code fragment assumes that all arguments
' have been properly initialized.
Call D3DX8.SimplifyMesh(Mesh, ByVal D3DXbAdjacency.GetBufferPointer, VAttWeights,
Vweights, MinV, Opts )
```

D3DX8.SphereBoundProbe

#Determines if a ray intersects the volume of a sphere's bounding box.

object.**SphereBoundProbe(** _
Center **As D3DVECTOR**, _
Radius **As Single**, _
RayPosition **As D3DVECTOR**, _
Raydirection **As D3DVECTOR) As Boolean**

Parts

object
Object expression that resolves to a **D3DX8** object.

Center
D3DVECTOR type, specifying the center coordinate of the sphere.

Radius
Radius of the sphere.

RayPosition
D3DVECTOR type, specifying the origin coordinate of the ray.

Raydirection
D3DVECTOR type, specifying the direction of the ray.

Return Values

Returns TRUE if the ray intersects the volume of the sphere's bounding box. Otherwise, returns FALSE.

IDH_D3DX8.SphereBoundProbe_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DXERR_INVALIDMESH.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

D3DXSphereBoundProbe determines if the ray intersects the volume of the sphere's bounding box, not just the surface of the sphere.

D3DX8.TesselateMesh

#Tessellates a mesh.

object.**TesselateMesh(_**
MeshIn **As D3DXMesh**, **_**
AdjacencyArray **As Long**, **_**
NumSegs **As Long, _**
QuadraticInterpNormals **As Boolean) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DX8** object.

MeshIn
D3DXMesh object representing the mesh to tessellate.

AdjacencyArray
First element of an array of three **Long** values per face that specify the three neighbors for each face in the source mesh.

NumSegs
Number of segments per edge to tessellate.

QuadraticInterpNormals
Boolean, set this to true to specify quadratic interpolation for normals.

Return Values

D3DXMesh object representing the returned tessellated mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

IDH_D3DX8.TesselateMesh_graphicsd3dxvb

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Note

This function tessellates by using the n-Patch algorithm.

D3DX8.ValidMesh

#Validates a mesh.

object.**ValidMesh(_**
MeshIn **As D3DXMesh**, **_**
Adjacency **As Long) As Boolean**

Parts

object
Object expression that resolves to a **D3DX8** object.

MeshIn
D3DXMesh object representing the mesh to tessellate.

Adjacency
First element of an array of three **Long** values per face that specify the three neighbors for each face in the source mesh.

Return Values

Returns True if MehsIn is valid, otherwise returns False.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

D3DXERR_INVALIDMESH

E_OUTOFMEMORY

IDH_D3DX8.ValidMesh_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim Mesh As D3DXMesh
Dim D3DXbAdjacency As D3DXBuffer

' This code fragment assumes that all arguments
' have been properly initialized.
Call D3DX8.GeneratePMesh(Mesh, ByVal D3DXbAdjacency.GetBufferPointer)
```

D3DXBaseMesh

#Applications use the methods of the **D3DXBaseMesh** class to manipulate and query mesh and progressive mesh objects.

The methods of the **D3DXBaseMesh** class can be organized into the following groups.

Buffers	**GetIndexBuffer**
	GetVertexBuffer
	LockIndexBuffer
	LockVertexBuffer
	UnlockIndexBuffer
	UnlockVertexBuffer
Copying	**CloneMesh**
	CloneMeshFVF
Faces	**GetNumFaces**
Information	**GetDevice**
	GetOptions
Rendering	**DrawSubset**
	GetAttributeTable
Vertices	**GetDeclaration**

IDH_D3DXBaseMesh_graphicsd3dxvb

GetFVF

GetNumVertices

See Also

D3DXMesh, **D3DXPMesh**

D3DXBaseMesh.CloneMesh

#Clones a mesh using a declarator.

object.**CloneMesh(** _
Options **As Long**, _
Declaration **As Any**, _
Device **As Direct3DDevice8) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

Declaration
First element of an array of **Long** values, representing the declarator to describe the vertex format of the vertices in the output mesh.

Device
Direct3DDevice8 object representing the device object associated with the mesh.

Return Values

D3DXMesh object, representing the cloned mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXBaseMesh.CloneMesh_graphicsd3dxvb

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

See Also

D3DX8.DeclaratorFromFVF, **D3DXBaseMesh.GetDeclaration**

D3DXBaseMesh.CloneMeshFVF

#Clones a mesh using a flexible vertex format (FVF) code.

```
object.CloneMeshFVF( _
    Options As Long, _
    FVF As Long, _
    Device As Direct3DDevice8) As D3DXMesh
```

Parts

object
: **Object** expression that resolves to a **D3DXBaseMesh** object.

Options
: A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

FVF
: Combination of flexible vertex format flags that specifies the vertex format for the vertices in the output mesh.

Device
: **Direct3DDevice8** object representing the device object associated with the mesh.

Return Values

D3DXMesh object, representing the cloned mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

IDH_D3DXBaseMesh.CloneMeshFVF_graphicsd3dxvb

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

Remarks

CloneMeshFVF can be used to convert a mesh from one FVF to another.

See Also

D3DX8.FVFFromDeclarator, **D3DXBaseMesh.GetFVF**

D3DXBaseMesh.DrawSubset

#Draws a subset of a mesh.

object.**DrawSubset(** _
attribId **As Long)**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

attribId
A **Long** value that specifies which subset of the mesh to draw. This value is used to differentiate faces in a mesh as belonging to one or more attribute groups.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXBaseMesh.DrawSubset_graphicsd3dxvb

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

Remarks

An attribute table is used to identify areas of the mesh that need to be drawn with different textures, render states, materials, and so on. In addition, the application can use the attribute table to hide portions of a mesh by not drawing a given attribute identifier (*attribId*) when drawing the frame.

See Also

D3DXBaseMesh.GetAttributeTable

D3DXBaseMesh.GetAttributeTable

#Retrieves either an attribute table for a mesh, or the number of entries stored in an attribute table for a mesh.

object.**GetAttributeTable(** _
AttribTable **As Any**, _
AttribTabSize **As Long)**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

AttribTable
First element of an array of **D3DXATTRIBUTERANGE** types, representing the entries in the mesh's attribute table. Specify ByVal 0 to retrieve the value for *AttribTabSize*.

AttribTabSize
Either the number of entries stored in *AttribTable*, or a value to be filled in with the number of entries stored in the attribute table for the mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

IDH_D3DXBaseMesh.GetAttributeTable_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

Remarks

An attribute table is created by **D3DXMesh.Optimize** and by passing D3DXMESHOPT_ATTRSORT for the *Flags* parameter.

An attribute table is used to identify areas of the mesh that need to be drawn with different textures, render states, materials, and so on. In addition, the application can use the attribute table to hide portions of a mesh by not drawing a given attribute identifier when drawing the frame.

D3DXBaseMesh.GetDeclaration

#Retrieves a declaration describing the vertices in the mesh.

object.**GetDeclaration(** _
Declaration **As Long)**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Declaration
First element of an array describing the vertex format of the vertices in the queried mesh. The upper limit of this declarator array is MAX_FVF_DECL_SIZE, limiting the declarator to a maximum of 15 **Long** values. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXBaseMesh.GetDeclaration_graphicsd3dxvb

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

Remarks

This code fragment shows how to pass in a D3DXDECLARATOR to retrieve the declaration of a mesh. This assumes there is an existing D3DXMesh object (d3dxm) that contains a mesh.

```
Dim d3dxd As D3DXDECLARATOR

'This assumes that (d3dxm) has been properly initialized
Call d3dxd.GetDeclaration(d3dxd.Value(0))
```

See Also

D3DXBaseMesh.GetFVF

D3DXBaseMesh.GetDevice

#Retrieves the device associated with the mesh.

object.**GetDevice() As Direct3DDevice8**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Return Values

Direct3DDevice8 object, identifying the Microsoft® Direct3D® device associated with the mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXBaseMesh.GetDevice_graphicsd3dxvb

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

D3DXBaseMesh.GetFVF

#Retrieves the flexible vertex format of the vertices in the mesh.

object.**GetFVF() As Long**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Return Values

Returns a combination of flexible vertex format flags that describes the vertex format of the vertices in the queried mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

See Also

D3DXBaseMesh.GetDeclaration

IDH_D3DXBaseMesh.GetFVF_graphicsd3dxvb

D3DXBaseMesh.GetIndexBuffer

#Retrieves the data in an index buffer.

object.**GetIndexBuffer() As Direct3DIndexBuffer8**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Return Values

Direct3DIndexBuffer8 object representing the index buffer object associated with the mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

D3DXBaseMesh.GetNumFaces

#Retrieves the number of faces in the mesh.

object.**GetNumFaces() As Long**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

IDH_D3DXBaseMesh.GetIndexBuffer_graphicsd3dxvb
IDH_D3DXBaseMesh.GetNumFaces_graphicsd3dxvb

Return Values

Returns the number of faces in the mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

D3DXBaseMesh.GetNumVertices

#Retrieves the number of vertices in the mesh.

object.**GetNumVertices() As Long**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Return Values

Returns the number of vertices in the mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXBaseMesh.GetNumVertices_graphicsd3dxvb

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

D3DXBaseMesh.GetOptions

#Retrieves the mesh options enabled for this mesh at creation time.

object.**GetOptions() As Long**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Return Values

Returns a combination of one or more of the flags defined by the **CONST_D3DXMESH** enumeration, indicating the options enabled for this mesh at creation time.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

D3DXBaseMesh.GetVertexBuffer

#Retrieves a vertex buffer.

IDH_D3DXBaseMesh.GetOptions_graphicsd3dxvb
IDH_D3DXBaseMesh.GetVertexBuffer_graphicsd3dxvb

object.**GetVertexBuffer() As Direct3DVertexBuffer8**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Return Values

Direct3DVertexBuffer8 object representing the vertex buffer object associated with the mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

D3DXBaseMesh.LockIndexBuffer

#Locks an index buffer and obtains access to the index buffer memory.

object.**LockIndexBuffer(_**
Flags **As Long) As Long**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Flags
Combination of one or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing how the index buffer memory should be locked.

IDH_D3DXBaseMesh.LockIndexBuffer_graphicsd3dxvb

Return Values

Returns a **Long** value providing access to the locked index data.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

Remarks

When working with index buffers, you are allowed to make multiple lock calls. However, you must ensure that the number of lock calls matches the number of unlock calls. **DrawPrimitive** calls do not succeed with any outstanding lock count on any currently set index buffer.

See Also

D3DXBaseMesh.UnlockIndexBuffer

D3DXBaseMesh.LockVertexBuffer

#Locks a vertex buffer and obtains access to the vertex buffer memory.

object.**LockVertexBuffer(_**
Flags **As Long) As Long**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Flags
Combination of one or more valid locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, describing how the vertex buffer memory should be locked.

IDH_D3DXBaseMesh.LockVertexBuffer_graphicsd3dxvb

Return Values

Returns a **Long** value providing access to the locked vertex data.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

Remarks

When working with vertex buffers, you are allowed to make multiple lock calls. However, you must ensure that the number of lock calls match the number of unlock calls. **DrawPrimitive** calls do not succeed with any outstanding lock count on any currently set vertex buffer.

See Also

D3DXBaseMesh.UnlockVertexBuffer

D3DXBaseMesh.UnlockIndexBuffer

#Unlocks an index buffer.

object.**UnlockIndexBuffer()**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

IDH_D3DXBaseMesh.UnlockIndexBuffer_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

See Also

D3DXBaseMesh.LockIndexBuffer

D3DXBaseMesh.UnlockVertexBuffer

#Unlocks a vertex buffer.

object.**UnlockVertexBuffer()**

Parts

object
Object expression that resolves to a **D3DXBaseMesh** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Applies To

This method applies to the following classes, which implement methods from **D3DXBaseMesh**.

- **D3DXMesh**
- **D3DXPMesh**

IDH_D3DXBaseMesh.UnlockVertexBuffer_graphicsd3dxvb

See Also

D3DXBaseMesh.LockVertexBuffer

D3DXBuffer

#The **D3DXBuffer** class is used as a data buffer that stores vertex, adjacency, and material information during mesh optimization and loading operations. The buffer object is used to return arbitrary length data.

Also, buffer objects are used to return object code and error messages in methods that assemble vertex and pixel shaders.

The **D3DXBuffer** class is obtained by calling the **D3DX8.CreateBuffer** method.

The methods of the **D3DXBuffer** class can be organized into the following group.

Information	**GetBufferPointer**
	GetBufferSize

D3DXBuffer.GetBufferPointer

#Retrieves a handle to the data in the buffer.

object.**GetBufferPointer() As Long**

Parts

object
Object expression that resolves to a **D3DXBuffer** object.

Return Values

Returns a handle to the data in the buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXBuffer_graphicsd3dxvb
IDH_D3DXBuffer.GetBufferPointer_graphicsd3dxvb

See Also

D3DX8.BufferGetData, **D3DX8.BufferGetMaterial**, **D3DX8.BufferGetTextureName**, **D3DX8.BufferSetData**

D3DXBuffer.GetBufferSize

#Retrieves the total size of the data in the buffer.

object.**GetBufferSize() As Long**

Parts

object
Object expression that resolves to a **D3DXBuffer** object.

Return Values

Returns the total size of the data in the buffer, in bytes.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXFont

#The **D3DXFont** class is used to encapsulate the textures and resources needed to render a specific font on a specific device.

The **D3DXFont** class is obtained by calling the **D3DX8.CreateFont** method.

The methods of the **D3DXFont** class can be organized into the following groups.

Drawing	**Begin**
	DrawTextW
	End
Information	**GetDevice**
	GetLogFont

IDH_D3DXBuffer.GetBufferSize_graphicsd3dxvb
IDH_D3DXFont_graphicsd3dxvb

D3DXFont.Begin

#Prepares a device for drawing text.

object.**Begin()**

Parts

object
Object expression that resolves to a **D3DXFont** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method is optional. If a **DrawTextW** method is called outside of a **Begin** and **D3DXFont.End** pair, then Microsoft® Direct3D® calls **Begin** and **End**.

D3DXFont.DrawTextW

#Draws formatted text on a Microsoft® Direct3D® device.

object.**DrawTextW(_**
Text **As String**, _
Count **As Long**, _
DestRECT **As RECT**, _
Format **As Long**, _
Color **As Long) As Long**

Parts

object
Object expression that resolves to a **D3DXFont** object.

Text

IDH_D3DXFont.Begin_graphicsd3dxvb
IDH_D3DXFont.DrawTextW_graphicsd3dxvb

The string to draw.

Count
Specifies the number of characters in the string. If Count is –1, then **DrawTextW** will compute the character count automatically. -1 might not be specified if *Format* includes DT_MODIFYSTRING.

DestRECT
RECT type that contains the rectangle, in logical coordinates, in which the text is to be formatted.

Format
A combinations of values from the CONST_DTFLAGS enumeration, specifying the text formatting. If *Format* includes DT_MODIFYSTRING, the function could add up to four additional characters to this string. The buffer containing the string should be large enough to accommodate these extra characters.

Color
D3DCOLORVALUE type, specifying the color of the text.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_NOTAVAILABLE

D3DERR_OUTOFVIDEOMEMORY

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **DrawTextW** method uses the device context's selected font, text color, and background color to draw the text. Unless the DT_NOCLIP format is used, **DrawText** clips the text so that it does not appear outside the specified rectangle. All formatting is assumed to have multiple lines unless the DT_SINGLELINE format is specified.

If the selected font is too large for the rectangle, the **DrawTextW** method does not attempt to substitute a smaller font.

The **DrawTextW** method supports only fonts whose escapement and orientation are both zero.

D3DXFont.End

#Restores the device state to how it was when **D3DXFont.Begin** was called.

object.**End()**

Parts

object
Object expression that resolves to a **D3DXFont** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXFont.Begin

D3DXFont.GetDevice

#Retrieves the Microsoft® Direct3D® device associated with the font object.

object.**GetDevice() As Direct3DDevice8**

Parts

object
Object expression that resolves to a **D3DXFont** object.

Return Values

A **Direct3DDevice8** object, representing the Direct3D device object associated with the font object.

IDH_D3DXFont.End_graphicsd3dxvb
IDH_D3DXFont.GetDevice_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXFont.GetLogFont

#Retrieves the attributes of the font.

object.**GetLogFont(***LogFont* **As LOGFONT)**

Parts

object
Object expression that resolves to a **D3DXFont** object.

LogFont
LOGFONT type, to be filled with the attributes of the font.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXMesh

#Applications use the methods of the **D3DXMesh** class to manipulate mesh objects.

The **D3DXMesh** class is obtained by calling the **D3DX8.CreateMesh** or **D3DX8.CreateMeshFVF** methods.

The **D3DXMesh** class implements the following **D3DXBaseMesh** methods, which can be organized into the following groups.

Buffers	**GetIndexBuffer**
	GetVertexBuffer
	LockIndexBuffer
	LockVertexBuffer
	UnlockIndexBuffer

IDH_D3DXFont.GetLogFont_graphicsd3dxvb
IDH_D3DXMesh_graphicsd3dxvb

	UnlockVertexBuffer
Copying	**CloneMesh**
	CloneMeshFVF
Faces	**GetNumFaces**
Information	**GetDevice**
	GetOptions
Rendering	**DrawSubset**
	GetAttributeTable
Vertices	**GetDeclaration**
	GetFVF
	GetNumVertices

The methods of the **D3DXMesh** class can be organized into the following groups.

Locking	**LockAttributeBuffer**
	UnlockAttributeBuffer
Miscellaneous	**ConvertAdjacencyToPointReps**
	ConvertPointRepsToAdjacency
	GenerateAdjacency
Optimization	**Optimize**
	OptimizeInplace

D3DXMesh.ConvertAdjacencyToPointReps

[#]Generates point representatives for the mesh.

object.**ConvertAdjacencyToPointReps(_**
Adjacency **As Any**, _
PointRep **As Any)**

Parts

object
Object expression that resolves to a **D3DXMesh** object.

Adjacency

IDH_D3DXMesh.ConvertAdjacencyToPointReps_graphicsd3dxvb

First element of an array of **Long** values, representing the destination buffer for the face adjacency array of the mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh. This size of this array is the maximum number of faces multiplied by 3. See Remarks

PointRep

First element of an array of **Long** values, representing the point representatives for the mesh. The point representatives are stored as an array of indices with one element per vertex. The size of this array is the number of vertices in the mesh. You can specify ByVal 0 for this parameter, if the vertices do not need to be relabeled.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDMESH

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim D3DXbAdjacency As D3DXBuffer
Dim PointRep As Any

' This code fragment assumes that D3DXbAdjacencyand PointRep have been properly
' initialized.
Call D3DX8.ConvertPointRepsToAdjacency(ByVal D3DXbAdjacency.GetBufferPointer,
PointRep)
```

Remarks

Point representatives are a method of describing mesh adjacency by fusing two vertices with the same x-, y-, and z- coordinates but with different normal coordinates, u- and v-coordinates, color data, and so on. In a perfectly smooth mesh (a mesh without creases, boundaries, or holes), point representatives are redundant. In that case, it is possible to compute adjacent triangles by using a hash table to locate edges that share vertex indices. In the case of creases and texture boundaries, the

point representatives are required to give a unique vertex index to use in a hash table for a group of co-located vertices that make up a vertex in the mesh.

See Also

D3DXMesh.ConvertPointRepsToAdjacency

D3DXMesh.ConvertPointRepsToAdjacency

[#]Converts point representative data stored in Microsoft® DirectX® (.x) files to face adjacency information that is more flexible for optimization and simplification operations.

object.**ConvertPointRepsToAdjacency(** _
PointRep **As Any**, _
FaceAdjacency **As Any)**

Parts

object
Object expression that resolves to a **D3DXMesh** object.

PointRep
First element of an array of **Long** values, representing the point representatives for the mesh. The point representatives are stored as an array of indices with one element per vertex. The size of this array is the number of vertices in the mesh. You can specify ByVal 0 for this parameter, if the vertices do not need to be relabeled.

FaceAdjacency
First element of an array of **Long** values, representing the destination buffer for the face adjacency array of the mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh. This size of this array is the maximum number of faces multiplied by 3. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

[#] IDH_D3DXMesh.ConvertPointRepsToAdjacency_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Point representatives are a method of describing mesh adjacency by fusing two vertices with the same x-, y-, and z- coordinates but with different normal coordinates, u- and v-coordinates, color data, an so on. In a perfectly smooth mesh (a mesh without creases, boundaries, or holes), point representatives are redundant. In that case, it is possible to compute adjacent triangles by using a hash table to locate edges that share vertex indices. In the case of creases and texture boundaries, the point representatives are required to give a unique vertex index to use in a hash table for a group of co-located vertices that make up a vertex in the mesh.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim d3dxbAdjacency As D3DXBuffer
Dim PointRep As Any

' This code fragment assumes that d3dxbAdjacency and PointRep have been properly
' initialized.
Call D3DX8.ConvertPointRepsToAdjacency(ByVal PointRep.GetBufferPointer, ByVal
d3dxbAdjacency.GetBufferPointer)
```

See Also

D3DXMesh.ConvertAdjacencyToPointReps

D3DXMesh.GenerateAdjacency

#Converts point representative data stored in Microsoft® DirectX® (.x) files to face adjacency information that is more flexible for optimization and simplification operations.

object.**GenerateAdjacency(** _
Epsilon **As Single**, _
Adjacency **As Any)**

Parts

object
Object expression that resolves to a **D3DXMesh** object.

Epsilon
Separation distance under which vertices are welded. This parameter is currently ignored and uses an epsilon of 0.0.

IDH_D3DXMesh.GenerateAdjacency_graphicsd3dxvb

Adjacency
First element of an array of **Long** values, representing the destination buffer for the face adjacency array of the mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh. This size of this array is the maximum number of faces multiplied by 3. See Remarks.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method only logically welds the vertices and no changes are made to the mesh.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim d3dxbAdjacency As D3DXBuffer
Dim Eps As Single

' This code fragment assumes that d3dxbAdjacency and Eps have been properly
' initialized.
Call D3DX8.GenerateAdjacency(Eps, ByVal d3dxbAdjacency.GetBufferPointer)
```

D3DXMesh.LockAttributeBuffer

#Locks an attribute buffer.

object.**LockAttributeBuffer(_**
Flags **As Long) As Long**

Parts

object
Object expression that resolves to a **D3DXMesh** object.

IDH_D3DXMesh.LockAttributeBuffer_graphicsd3dxvb

Flags
Combination of one or more locking flags defined by the **CONST_D3DLOCKFLAGS** enumeration, indicating how the attribute buffer memory should be locked.

Return Values

First element of an array of **Long** values, filled with the returned attribute buffer data.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMesh.UnlockAttributeBuffer, **D3DXATTRIBUTERANGE**

D3DXMesh.Optimize

#Controls the reordering of mesh faces and vertices to optimize performance.

object.**Optimize(** _
Flags **As Any**, _
Adjacency **As Any**, _
OptAdj **As Any**, _
FaceRemap **As Any**, _
BuffVertexRemap **As D3DXBuffer**, _
OptMesh **As D3DXMesh)**

Parts

object
Object expression that resolves to a **D3DXMesh** object.

Flags
A combination of one or more flags defined by the **CONST_D3DXMESHOPT** enumeration, specifying the type of optimization to perform.

Note that the D3DXMESHOPT_STRIPREORDER and D3DXMESHOPT_VERTEXCACHE optimization flags are mutually exclusive.

Adjacency
First element of an array of three **Long** values per face that specify the three neighbors for each face in the source mesh.

IDH_D3DXMesh.Optimize_graphicsd3dxvb

OptAdj
First element of an array, representing the destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

FaceRemap
First element of an array, representing the destination buffer containing the new index for each face.

BuffVertexRemap
D3DXBuffer object, containing the new index for each vertex.

OptMesh
D3DXMesh object, representing the optimized mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method is very similar to the **D3DXBaseMesh.CloneMesh** method, except that it can perform optimization while generating the new clone of the mesh.

The output mesh inherits all of the creation parameters of the input mesh.

D3DXMesh.OptimizeInplace

#Controls the reordering of mesh faces and vertices to optimize performance.

object.**OptimizeInplace(** _
Flags **As Long**, _
AdjacencyIn **As Any**, _
AdjacencyOut **As Any**, _
FaceRemap **As Any**, _
VertexRemapOut **As D3DXBuffer)**

Parts

object
Object expression that resolves to a **D3DXMesh** object.

Flags

IDH_D3DXMesh.OptimizeInplace_graphicsd3dxvb

A combination of one or more flags defined by the **CONST_D3DXMESHOPT** enumeration, specifying the type of optimization to perform.

Note that the D3DXMESHOPT_STRIPREORDER and D3DXMESHOPT_VERTEXCACHE optimization flags are mutually exclusive.

AdjacencyIn

First element of an array of three **Long** values per face that specify the three neighbors for each face in the source mesh.

AdjacencyOut

First element of an array, representing the destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

FaceRemap

First element of an array, representing the destination buffer containing the new index for each face.

VertexRemapOut

D3DXBuffer object, containing the new index for each vertex.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_CANNOTATTRSORT

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The output mesh inherits all of the creation parameters of the input mesh.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim Opts As Long
Dim D3DXbAdjacencyIn As D3DXBuffer
Dim D3DXbAdjacencyOut As Any
Dim FaceRemap As Any
Dim FaceRemap As D3DXBuffer

' This code fragment assumes that all arguments
' have been properly initialized.
```

```
Call D3DX8.OptimizeInPlace( Opts, ByVal D3DXbAdjacencyIn.GetBufferPointer,
D3DXbAdjacencyOut, _
                                FaceRemap, FaceRemap )
```

Note

This method fails if the mesh is sharing its vertex buffer with another mesh, unless the D3DXMESHOPT_IGNOREVERTS flag is set in the *Flags* parameter.

D3DXMesh.UnlockAttributeBuffer

#Unlocks an attribute buffer.

object.**UnlockAttributeBuffer()**

Parts

object
Object expression that resolves to a **D3DXMesh** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMesh.LockAttributeBuffer

D3DXPMesh

#Applications use the methods of the **D3DXPMesh** class to manipulate progressive mesh objects. A progressive mesh enables progressive refinement of the mesh object.

The **D3DXPMesh** class is obtained by calling the **D3DX8.GeneratePMesh** method.

The **D3DXPMesh** class inherits the following **D3DXBaseMesh** methods, which can be organized into the following groups.

Buffers	**GetIndexBuffer**
	GetVertexBuffer
	LockIndexBuffer

IDH_D3DXMesh.UnlockAttributeBuffer_graphicsd3dxvb
IDH_D3DXPMesh_graphicsd3dxvb

	LockVertexBuffer
	UnlockIndexBuffer
	UnlockVertexBuffer
Copying	**CloneMesh**
	CloneMeshFVF
Faces	**GetNumFaces**
Information	**GetDevice**
	GetOptions
Rendering	**DrawSubset**
	GetAttributeTable
Vertices	**GetDeclaration**
	GetFVF
	GetNumVertices

The methods of the **D3DXPMesh** class can be organized into the following groups.

Copying	**ClonePMesh**
	ClonePMeshFVF
Faces	**GetMaxFaces**
	GetMinFaces
	SetNumFaces
Miscellaneous	**GetAdjacency**
	Optimize
Vertices	**GetMaxVertices**
	GetMinVertices
	SetNumVertices

D3DXPMesh.ClonePMesh

#Clones a progressive mesh using a declarator.

object.**ClonePMesh(** _
Options **As Long**, _
Declaration **As Any**, _
Device **As Direct3DDevice8) As D3DXPMesh**

IDH_D3DXPMesh.ClonePMesh_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

Declaration
First element of an array of **Long** values, representing the declarator to describe the vertex format of the vertices in the output mesh. This parameter must map directly to an FVF.

Device
Direct3DDevice8 object representing the device object associated with the mesh.

Return Values

D3DXPMesh object, representing the cloned progressive mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DX8.DeclaratorFromFVF, **D3DXBaseMesh.GetDeclaration**

D3DXPMesh.ClonePMeshFVF

#Clones a progressive mesh using a flexible vertex format (FVF) code.

object.**ClonePMeshFVF(** _
Options **As Long**, _
FVF **As Long**, _
Device **As Direct3DDevice8) As D3DXPMesh**

IDH_D3DXPMesh.ClonePMeshFVF_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

FVF
Combination of flexible vertex format flags that specifies the vertex format for the vertices in the output mesh.

Device
Direct3DDevice8 object representing the device object associated with the mesh.

Return Values

D3DXPMesh object, representing the cloned progressive mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

ClonePMeshFVF can be used to convert a progressive mesh from one FVF to another.

See Also

D3DX8.FVFFromDeclarator, **D3DXBaseMesh.GetFVF**

D3DXPMesh.GetAdjacency

#Returns the face adjacency array of the mesh.

object.**GetAdjacency(_**
AdjacencyOut **As Any**

IDH_D3DXPMesh.GetAdjacency_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

AdjacencyOut
Returned face adjacency array of the mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh. The size of the adjacency array is the maximum number of faces multiplied by 3.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXPMesh.GetMaxFaces

D3DXPMesh.GetMaxFaces

#Retrieves the maximum number of faces that the progressive mesh supports.

object.**GetMaxFaces() As Long**

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

Return Values

Returns the maximum number of faces supported by the progressive mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXPMesh.GetMaxFaces_graphicsd3dxvb

D3DXPMesh.GetMaxVertices

#Retrieves the maximum number of vertices that the progressive mesh supports.

object.**GetMaxVertices() As Long**

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

Return Values

Returns the maximum number of vertices supported by the progressive mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXPMesh.GetMinFaces

#Retrieves the minimum number of faces that the progressive mesh supports.

object.**GetMinFaces() As Long**

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

Return Values

Returns the minimum number of faces supported by the progressive mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXPMesh.GetMaxVertices_graphicsd3dxvb
IDH_D3DXPMesh.GetMinFaces_graphicsd3dxvb

D3DXPMesh.GetMinVertices

#Retrieves the minimum number of vertices that the progressive mesh supports.

object.**GetMinVertices() As Long**

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

Return Values

Returns the minimum number of vertices supported by the progressive mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXPMesh.Optimize

#Controls the reordering of mesh faces and vertices to optimize performance, generating an output mesh.

object.**Optimize(** _
Flags **As Long**, _
AdjacencyOut **As Any**, _
FaceRemap **As Any**, _
VertexRemapOut **As D3DXBuffer) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

Flags
A combination of one or more flags defined by the **CONST_D3DXMESHOPT** enumeration, specifying the type of optimization to perform.

Note that the D3DXMESHOPT_STRIPREORDER and D3DXMESHOPT_VERTEXCACHE optimization flags are mutually exclusive.

AdjacencyOut

IDH_D3DXPMesh.GetMinVertices_graphicsd3dxvb
IDH_D3DXPMesh.Optimize_graphicsd3dxvb

A destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh. The size of this array is the maximum number of faces multiplied by 3.

FaceRemap
Destination buffer containing the new index for each face.

VertexRemapOut
D3DXBuffer object; containing the new index for each vertex.

Return Values

D3DXMesh object, representing the optimized mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

ClonePMeshFVF can be used to convert a progressive mesh from one FVF to another.

See Also

D3DX8.FVFFromDeclarator, **D3DXBaseMesh.GetFVF**

D3DXPMesh.SetNumFaces

#Sets the current level of detail to as close to the specified number of faces as possible.

object.**SetNumFaces(** _
Faces **As Long)**

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

IDH_D3DXPMesh.SetNumFaces_graphicsd3dxvb

Faces
Target number of faces. This value specifies the desired change in the level of detail (LOD).

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If the number of faces is greater than the maximum number of faces, it is capped at the maximum number of faces returned by **D3DXPMesh.GetMaxFaces**. If the number of faces is less than the minimum number of faces, it is capped at the minimum number of faces returned by **D3DXPMesh.GetMinFaces**.

The number of faces after this call might be off by one because some edge collapse might introduce or remove one face or two. For example, if you try setting the number of faces to an intermediate value such as 5, when 4 and 6 are possible, 4 is always the result.

D3DXPMesh.SetNumVertices

#Sets the current level of detail (LOD) to as close to the specified number of vertices as possible.

object.**SetNumVertices(** _
Vertices **As Long)**

Parts

object
Object expression that resolves to a **D3DXPMesh** object.

Vertices
Target number of vertices. This value specifies the desired change in the LOD.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXPMesh.SetNumVertices_graphicsd3dxvb

Remarks

If the number of vertices is greater than the maximum number of vertices, it is capped at the maximum number of vertices returned by **D3DXPMesh.GetMaxVertices**. If the number of vertices is less than the minimum number of vertices, it is capped at the minimum number of vertices returned by **D3DXPMesh.GetMinVertices**.

The number of vertices after this call might be off by one because some edge collapse might introduce or remove one face or two. For example, if you try setting the number of faces to an intermediate value such as 5, when 4 and 6 are possible, 4 is always the result.

D3DXRenderToSurface

#The **D3DXRenderToSurface** class is used to generalize the process of rendering to surfaces. If the surface is not a render target, a compatible render target is used, and the result is copied to the surface at the end of the scene.

The **D3DXRenderToSurface** class is obtained by calling the **D3DX8.CreateRenderToSurface** method.

The methods of the **D3DXRenderToSurface** class can be organized into the following groups.

Information	**GetDesc**
	GetDevice
Rendering	**BeginScene**
	EndScene

D3DXRenderToSurface.BeginScene

#Begins a scene.

object.**BeginScene(** _
Surface **As Direct3DSurface8**, _
Viewport **As D3DVIEWPORT8)**

Parts

object
Object expression that resolves to a **D3DXRenderToSurface** object.

Surface
A **Direct3DSurface8** object, representing the render surface.

IDH_D3DXRenderToSurface_graphicsd3dxvb
IDH_D3DXRenderToSurface.BeginScene_graphicsd3dxvb

Viewport
D3DVIEWPORT8 type, describing the viewport for the scene.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

D3DXERR_INVALIDDATA

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXRenderToSurface.EndScene

D3DXRenderToSurface.EndScene

#Ends a scene.

object.**EndScene()**

Parts

object
Object expression that resolves to a **D3DXRenderToSurface** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXRenderToSurface.EndScene_graphicsd3dxvb

See Also

D3DXRenderToSurface.BeginScene

D3DXRenderToSurface.GetDesc

#Retrieves the parameters of the render surface.

object.**GetDesc(_**
Parameters **As D3DXRTS_DESC)**

Parts

object
Object expression that resolves to a **D3DXRenderToSurface** object.

Parameters
D3DXRTS_DESC type, describing the parameters of the render surface.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXRenderToSurface.GetDevice

#Retrieves the Microsoft® Direct3D® device associated with the render surface.

object.**GetDevice() As Direct3DDevice8**

Parts

object
Object expression that resolves to a **D3DXRenderToSurface** object.

Return Values

A **Direct3DDevice8** object, representing the Direct3D device object associated with the render surface.

IDH_D3DXRenderToSurface.GetDesc_graphicsd3dxvb
IDH_D3DXRenderToSurface.GetDevice_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSkinMesh

#Applications use the methods of the **D3DXSkinMesh** class to manipulate skin mesh objects.

The **D3DXSkinMesh** class is obtained by calling one of the following methods.

- **D3DX8.CreateSkinMesh**
- **D3DX8.CreateSkinMeshFromMesh**
- **D3DX8.CreateSkinMeshFVF**

To load a skin mesh from a Microsoft® DirectX® (.x) file data object, use the **D3DX8.LoadSkinMeshFromXof** method.

Skinning is the technique of having different vertices of a mesh be transformed by multiple transform matrices and the results of the vertex being transformed with each individual matrix blended together to obtain the final transformed vertex.

This process is commonly called skinning because it is used to deform a mesh defining a skin of some character based on the animation of a skeleton of bones. In the simplest case of rigid animation, different parts of the character mesh are associated with a unique skeleton bone and vertices are transformed using the transform of that bone.

When discussing skinning, it is useful to define the following terms.

- *Skin*. The triangle mesh being rendered using deformations. Typically this is the model of the character in some convenient pose.
- *Bone*. A transform matrix that affects one or more vertices of the skin.
- *Skeleton*. A hierarchy of all the bones that affect a skin mesh.
- *Pose*. A set of bone transforms that completely defines the transforms for all the bones that affect a given skin mesh.
- *Initial Pose*. This is the pose in which the mesh was associated with the skeleton. When the skeleton is in this pose, the deformed mesh is identical to the original undeformed mesh.
- *Bone Space Transform*. This transform transforms the mesh into the local space of a particular bone.

IDH_D3DXSkinMesh_graphicsd3dxvb

- *Bone Weight*. The amount of influence a bone has on a given vertex. A weight of 1 means that the vertex is only affected by that bone and 0 means it is unaffected by the bone. The sum of weights of all the bones that affect a vertex should add up to 1 for all vertices.

A skinned character is defined by a set of meshes and a set of bones that affect the vertices of the meshes. The bones are represented as a transform hierarchy. For each mesh, there is a matrix for every bone that affects it that transforms the mesh into the local coordinate space of the bone. This matrix is the bone space transform of the bone for the mesh. This is defined at the time the skeleton is associated with the mesh in the authoring process.

The methods of the **D3DXSkinMesh** class can be organized into the following groups.

Buffers	**GetIndexBuffer**
	GetVertexBuffer
Conversion	**ConvertToBlendedMesh**
	ConvertToIndexedBlendedMesh
Information	**GetBoneInfluence**
	GetDevice
	GetMaxFaceInfluences
	GetMaxVertexInfluences
	GetNumBoneInfluences
	GetNumBones
Locking	**LockAttributeBuffer**
	LockIndexBuffer
	LockVertexBuffer
	UnlockAttributeBuffer
	UnlockIndexBuffer
	UnlockVertexBuffer
Miscellaneous	**GenerateSkinnedMesh**
	GetOriginalMesh
	SetBoneInfluence
	UpdateSkinnedMesh

D3DXSkinMesh.ConvertToBlendedMesh

#Returns a blended mesh.

object.**ConvertToBlendedMesh(** _
Options **As Long**, _
AdjacencyIn **As Any**, _
AdjacencyOut **As Any**, _
RetNumBoneCombinations **As Long**, _
RetBoneCombinationTable **As D3DXBuffer) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying options for the mesh.

AdjacencyIn
First element of an array of three **Long** values per face that specify the three neighbors for each face in the source mesh. See Remarks.

AdjacencyOut
First element in a destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh. See Remarks.

RetNumBoneCombinations
Number of entries.

RetBoneCombinationTable
First element in an array of **D3DXBONECOMBINATION** types. The count of types is *NumBoneCombinations*. Each bone combination table defines the four bones that you can draw at a time.

Return Values

Returns a **D3DXMesh** object, representing the blended mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXSkinMesh.ConvertToBlendedMesh_graphicsd3dxvb

Remarks

This method takes a skin mesh and converts it into a blended mesh (a mesh with bone influences) using the Microsoft® Direct3D® vertex blending functionality available in Microsoft DirectX® 7.*x*. For more information, see Geometry Blending.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim Opts As Long
Dim D3DXbAdjacencyIn As D3DXBuffer
Dim D3DXbAdjacencyOut As Any
Dim NumBC As Long
Dim RetTable As D3DXBuffer

' This code fragment assumes that all arguments
' have been properly initialized.
Call D3DX8.ConvertToBlendedMesh(Opts, ByVal D3DXbAdjacencyIn.GetBufferPointer,
D3DXbAdjacencyOut, _
                                NumBC, RetTable )
```

D3DXSkinMesh.ConvertToIndexedBlendedMesh

#Returns an indexed blended mesh.

object.**ConvertToIndexedBlendedMesh(** _
Options **As Long**, _
AdjacencyIn **As Any**, _
PaletteSize **As Long,** _
AdjacencyOut **As Any**, _
RetNumBoneCombinations **As Long**, _
RetBoneCombinationTable **As D3DXBuffer) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying options for the mesh.

AdjacencyIn
First element of an array of three **Long** values per face that specify the three neighbors for each face in the source mesh.

IDH_D3DXSkinMesh.ConvertToIndexedBlendedMesh_graphicsd3dxvb

PaletteSize
The palette size.

AdjacencyOut
A destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

RetNumBoneCombinations
Number of entries.

RetBoneCombinationTable
First element in an array of **D3DXBONECOMBINATION** types. The count of types is *NumBoneCombinations*. Each bone combination table defines the four bones that you can draw at a time.

Return Values

Returns a **D3DXMesh** object, representing the blended mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method takes a skin mesh and converts it into an indexed blended mesh using the Direct3D indexed vertex blending functionality available in DirectX 8.0. Indexed vertex blending uses indices to index into a matrix palette. For more information, see Geometry Blending. Converting a skin mesh into an indexed blended mesh enables you to render the mesh in a single drawing call.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim Opts As Long
Dim D3DXbAdjacencyIn As D3DXBuffer
Dim PalSize as Long
Dim D3DXbAdjacencyOut As Any
Dim NumBC As Long
Dim RetTable As D3DXBuffer

' This code fragment assumes that all arguments
' have been properly initialized.
```

```
Call D3DX8.ConvertToIndexedBlendedMesh(Opts, ByVal D3DXbAdjacencyIn.GetBufferPointer,
PalSize, D3DXbAdjacencyOut, _
                                NumBC, RetTable )
```

D3DXSkinMesh.GenerateSkinnedMesh

#Generates a skinned mesh.

object.**GenerateSkinnedMesh(_**
Options **As Long, _**
MinWeight **As Single, _**
AdjacencyIn **As Any, _**
AdjacencyOut **As Any) As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying options for the mesh.

MinWeight
Sets the minimum weight clamping value.

AdjacencyIn
First element of an array of three **Long** values per face that specify the three neighbors for each face in the source mesh. See Remarks.

AdjacencyOut
A destination buffer for the face adjacency array of the optimized mesh. The face adjacency is stored as an array of arrays. The innermost array is three indices of adjacent triangles, and the outer array is one set of face adjacency per triangle in the mesh.

Return Values

Returns a **D3DXMesh** object, representing the generated skinned mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXSkinMesh.GenerateSkinnedMesh_graphicsd3dxvb

Remarks

This method enables you to specify all matrices and apply them to each bone. **GenerateSkinnedMesh** does the geometry blending for you, so you do not need to used Direct3D to do the geometry blending. In particular, this method allows more that four bone influences per vertex.

The mesh generated by this method is the one that should be supplied to **D3DXSkinMesh.UpdateSkinnedMesh** each time the skeleton transforms are modified.

This method is useful for skin animations.

The following code fragment shows how to use a **D3DXBuffer** object to pass adjacency information.

```
Dim Opts As Long
Dim MinW As Single
Dim D3DXbAdjacencyIn As D3DXBuffer
Dim D3DXbAdjacencyOut As Any

' This code fragment assumes that all arguments
' have been properly initialized.
Call D3DX8.ConvertToIndexedBlendedMesh(Opts, MinW, ByVal
D3DXbAdjacencyIn.GetBufferPointer, PalSize, D3DXbAdjacencyOut)
```

D3DXSkinMesh.GetBoneInfluence

#Returns a list of values indicating how a bone effects the skin mesh.

object.**GetBoneInfluence(** _
Bone **As Long**, _
Vertices **As Long**, _
Weights **As Single**,**)**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Bone
Bone to query.

Vertices
The list of vertices affected by the bone.

Weights
List of weights associated with each vertex in the list of vertices affected by the bone.

IDH_D3DXSkinMesh.GetBoneInfluence_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSkinMesh.GetDevice

#Retrieves the device associated with the skin mesh.

object.**GetDevice() As Direct3DDevice8**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Return Values

Returns a **Direct3DDevice8** object, identifying the Microsoft® Direct3D® device associated with the mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSkinMesh.GetIndexBuffer

#Retrieves the data in an index buffer.

object.**GetIndexBuffer() As Direct3DIndexBuffer8**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

IDH_D3DXSkinMesh.GetDevice_graphicsd3dxvb
IDH_D3DXSkinMesh.GetIndexBuffer_graphicsd3dxvb

Return Values

A **Direct3DIndexBuffer8** object, representing the index buffer object associated with the skin mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSkinMesh.GetMaxFaceInfluences

#Return the maximum number of face influences.

object.**GetMaxFaceInfluences() As Long**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Return Values

Maximum number of influences effecting any single face in the skin mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Each face can be affected by *n* bones. The following diagram shows a face affected by 6 bones (the union of bones effecting each vertex of the face).

D3DXSkinMesh.GetMaxVertexInfluences

#Returns the maximum number of vertex influences.

IDH_D3DXSkinMesh.GetMaxFaceInfluences_graphicsd3dxvb
IDH_D3DXSkinMesh.GetMaxVertexInfluences_graphicsd3dxvb

object.**GetMaxVertexInfluences() As Long**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Return Values

Maximum number of influences effecting any single vertex in the skin mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Each vertex can be affected by *n* bones. If this method returns a value of 17, up to 17 bones effect a single vertex of the skin mesh. So, there are 17 separate weights on that vertex which add up to 1.

D3DXSkinMesh.GetNumBoneInfluences

#Returns the number of influences (weights) on a given bone.

object.**GetNumBoneInfluences(_**
Bone **As Long) As Long**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.
Bone
The bone to query.

Return Values

Returns the number of weights on the given bone. If *Bone* is an invalid bone number, **GetNumBoneInfluences** returns 0.

IDH_D3DXSkinMesh.GetNumBoneInfluences_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSkinMesh.GetNumBones

#Returns the number of bones in the skin mesh.

object.**GetNumBones() As Long**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Return Values

Number of bones in the skin mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSkinMesh.GetOriginalMesh

#Returns the skin mesh in its original (default) pose.

object.**GetOriginalMesh() As D3DXMesh**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Return Values

A **D3DXMesh** object, representing the skin mesh in its default pose.

IDH_D3DXSkinMesh.GetNumBones_graphicsd3dxvb
IDH_D3DXSkinMesh.GetOriginalMesh_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSkinMesh.GetVertexBuffer

#Retrieves the data in a vertex buffer.

object.**GetVertexBuffer() As Direct3DVertexBuffer8**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Return Values

Direct3DVertexBuffer8 object, representing the vertex buffer object associated with the skin mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSkinMesh.LockAttributeBuffer

#Locks an attribute buffer.

object.**LockAttributeBuffer(_**
Flags **As Long) As Long**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.
Flags

IDH_D3DXSkinMesh.GetVertexBuffer_graphicsd3dxvb
IDH_D3DXSkinMesh.LockAttributeBuffer_graphicsd3dxvb

A combination of one or more locking flags, indicating how the attribute buffer memory should be locked. Possible values are defined by the **CONST_D3DLOCKFLAGS** enumeration.

Return Values

First element of an array of **Long** values, filled with the returned attribute buffer data.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSkinMesh.LockIndexBuffer

#Locks an index buffer.

object.**LockIndexBuffer(_**
Flags **As Long) As Long**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Flags
A combination of one or more locking flags, describing how the index buffer memory should be locked. Possible values are defined by the **CONST_D3DLOCKFLAGS** enumeration.

Return Values

First element of an array of **BYTE** values, filled with the returned index data.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXSkinMesh.LockIndexBuffer_graphicsd3dxvb

Remarks

When working with index buffers, you are allowed to make multiple lock calls. However, you must ensure that the number of lock calls matches the number of unlock calls. **DrawPrimitive** calls do not succeed with any outstanding lock count on any currently set index buffer.

See Also

D3DXSkinMesh.UnlockIndexBuffer

D3DXSkinMesh.LockVertexBuffer

#Locks a vertex buffer.

object.**LockVertexBuffer(_**
Flags **As Long) As Long**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Flags
A combination of one or more locking flags, indicating how the vertex buffer memory should be locked. Possible values are defined by the **CONST_D3DLOCKFLAGS** enumeration.

Return Values

First element of an array of **BYTE** values, filled with the returned vertex data.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

When working with vertex buffers, you are allowed to make multiple lock calls. However, you must ensure that the number of lock calls match the number of unlock calls. **DrawPrimitive** calls do not succeed with any outstanding lock count on any currently set vertex buffer.

IDH_D3DXSkinMesh.LockVertexBuffer_graphicsd3dxvb

See Also

D3DXSkinMesh.UnlockVertexBuffer

D3DXSkinMesh.SetBoneInfluence

#Sets the bone influence for a bone in the skin mesh.

object.**SetBoneInfluence(** _
Bone **As Long,** _
NumInfluences **As Long,** _
Vertices **As Long,** _
Weights **As Single)**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Bone
Bone for which to set the influences.

NumInfluences
Number of influences to set for the bone.

Vertices
The list of vertices used to set the influences (weights).

Weights
The list of weights to set for the provided vertices. These values are used to programmatically fill in the skinning information for the skin mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to E_OUTOFMEMORY.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Using this method provides a mechanism to indicate how the given bone effects the given vertices. This method should be called for each bone that effects the mesh.

D3DXSkinMesh.UnlockAttributeBuffer

#Unlocks an attribute buffer.

IDH_D3DXSkinMesh.SetBoneInfluence_graphicsd3dxvb
IDH_D3DXSkinMesh.UnlockAttributeBuffer_graphicsd3dxvb

object.**UnlockAttributeBuffer()**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXSkinMesh.LockAttributeBuffer

D3DXSkinMesh.UnlockIndexBuffer

[#]Unlocks an index buffer.

object.**UnlockIndexBuffer()**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXSkinMesh.LockIndexBuffer

IDH_D3DXSkinMesh.UnlockIndexBuffer_graphicsd3dxvb

D3DXSkinMesh.UnlockVertexBuffer

#Unlocks a vertex buffer.

object.**UnlockVertexBuffer()**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXSkinMesh.LockVertexBuffer

D3DXSkinMesh.UpdateSkinnedMesh

#Updates a skinned mesh.

object.**UpdateSkinnedMesh(** _
BoneTransformsIn **As Any**, _
RetMesh **As D3DXMesh)**

Parts

object
Object expression that resolves to a **D3DXSkinMesh** object

BoneTransformsIn
First element of an array of **D3DMATRIX** types, representing the bone transforms. The length of this array is the count of all the bones in the skin mesh

RetMesh
A **D3DXMesh** object, to be filled with the generated skinned mesh.

Remarks

This method updates the vertices blended with the new bone transforms.

IDH_D3DXSkinMesh.UnlockVertexBuffer_graphicsd3dxvb
IDH_D3DXSkinMesh.UpdateSkinnedMesh_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXSkinMesh.GenerateSkinnedMesh

D3DXSPMesh

#Applications use the methods of the **D3DXSPMesh** class to manipulate simplification mesh objects. A simplification mesh is used to simplify a given mesh to a lower number of faces.

The **D3DXSPMesh** class is obtained by calling the **D3DX8.CreateSPMesh** method.

The methods of the **D3DXSPMesh** class can be organized into the following groups.

Copying	**CloneMesh**
	CloneMeshFVF
	ClonePMesh
	ClonePMeshFVF
Faces	**GetMaxFaces**
	GetNumFaces
	ReduceFaces
Information	**GetDevice**
	GetOptions
Vertices	**GetDeclaration**
	GetFVF
	GetMaxVertices
	GetNumVertices
	ReduceVertices

IDH_D3DXSPMesh_graphicsd3dxvb

D3DXSPMesh.CloneMesh

#Clones a mesh using a declarator.

object.**CloneMesh(** _
Options **As Long**, _
Declaration **As Any**, _
Device **As Direct3DDevice8,** _
AdjacencyOut **As Any**, _
VertexRemapOut **As Any) As D3DXPMesh**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

Declaration
First element of an array of **Long** values, representing the declarator to describe the vertex format of the vertices in the output mesh. This parameter must map directly to an FVF.

Device
Direct3DDevice8 object representing the device object associated with the mesh.

AdjacencyOut
First element of an array of three **Long** values per face that specify the three neighbors for each face in the source mesh

VertexRemapOut
First element of an array containing the index for each vertex.

Return Values

D3DXPMesh object, representing the cloned mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXSPMesh.CloneMesh_graphicsd3dxvb

See Also

D3DX8.DeclaratorFromFVF, **D3DXSPMesh.GetDeclaration**

D3DXSPMesh.CloneMeshFVF

#Clones a mesh using a flexible vertex format (FVF) code.

object.**CloneMeshFVF(** _
Options **As Long**, _
FVF **As Long**, _
Device **As Direct3DDevice8,** _
AdjacencyOut **As Any**, _
VertexRemapOut **As Any) As D3DXPMesh**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

FVF
Combination of flexible vertex format flags that specifies the vertex format for the vertices in the output mesh.

Device
Direct3DDevice8 object representing the device object associated with the mesh.

AdjacencyOut
First element of an array of three **Long** values per face that specify the three neighbors for each face in the source mesh.

VertexRemapOut
First element of an array containing the index for each vertex.

Return Values

D3DXPMesh object, representing the cloned mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

IDH_D3DXSPMesh.CloneMeshFVF_graphicsd3dxvb

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

CloneMeshFVF can be used to convert a mesh from one FVF to another.

See Also

D3DX8.FVFFromDeclarator, **D3DXSPMesh.GetFVF**

D3DXSPMesh.ClonePMesh

#Clones a progressive mesh using a declarator.

object.**ClonePMesh(** _
Options **As Long**, _
Declaration **As Any**, _
Device **As Direct3DDevice8,** _
VertexRemapOut **As Any) As D3DXPMesh**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

Declaration
First element of an array of **Long** values, representing the declarator to describe the vertex format of the vertices in the output mesh. This parameter must map directly to an FVF.

Device
Direct3DDevice8 object representing the device object associated with the mesh.

VertexRemapOut
First element of an array containing the index for each vertex.

Return Values

D3DXPMesh object, representing the cloned progressive mesh.

IDH_D3DXSPMesh.ClonePMesh_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_CANNOTATTRSORT

E_OUTOFMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DX8.DeclaratorFromFVF, **D3DXSPMesh.GetDeclaration**

D3DXSPMesh.ClonePMeshFVF

#Clones a progressive mesh using a flexible vertex format (FVF) code.

object.**ClonePMeshFVF(** _
Options **As Long**, _
FVF **As Long**, _
Device **As Direct3DDevice8,** _
VertexRemapOut **As Any) As D3DXPMesh**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Options
A combination of one or more flags defined by the **CONST_D3DXMESH** enumeration, specifying creation options for the mesh.

FVF
Combination of flexible vertex format flags that specifies the vertex format for the vertices in the output mesh.

Device
Direct3DDevice8 object representing the device object associated with the mesh.

VertexRemapOut
First element of an array containing the index for each vertex.

Return Values

D3DXPMesh object, representing the cloned progressive mesh.

IDH_D3DXSPMesh.ClonePMeshFVF_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

ClonePMeshFVF can be used to convert a progressive mesh from one FVF to another.

See Also

D3DX8.FVFFromDeclarator, **D3DXSPMesh.GetFVF**

D3DXSPMesh.GetDeclaration

#Retrieves a declaration describing the vertices in the mesh.

object.**GetDeclaration(** _
Declaration **As Long)**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Declaration
First element of an array describing the vertex format of the vertices in the queried mesh. The upper limit of this declarator array is MAX_FVF_DECL_SIZE, limiting the declarator to a maximum of 15 **Long** values.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXSPMesh.GetDeclaration_graphicsd3dxvb

See Also

D3DXSPMesh.GetFVF

D3DXSPMesh.GetDevice

#Retrieves the device object associated with the simplification mesh.

object.**GetDevice() As Direct3DDevice8**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Return Values

Direct3DDevice8 object, identifying the Microsoft® Direct3D® device associated with the simplification mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSPMesh.GetFVF

#Retrieves the flexible vertex format of the vertices in the simplification mesh.

object.**GetFVF() As Long**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Return Values

Combination of flexible vertex format flags that describe the vertex format of the vertices in the simplification mesh.

IDH_D3DXSPMesh.GetDevice_graphicsd3dxvb
IDH_D3DXSPMesh.GetFVF_graphicsd3dxvb

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXSPMesh.GetDeclaration

D3DXSPMesh.GetMaxFaces

#Retrieves the maximum number of faces that the simplification mesh supports.

object.**GetMaxFaces() As Long**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Return Values

Returns the maximum number of faces in the original mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSPMesh.GetMaxVertices

#Retrieves the maximum number of vertices that the simplification mesh supports.

object.**GetMaxVertices() As Long**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

IDH_D3DXSPMesh.GetMaxFaces_graphicsd3dxvb
IDH_D3DXSPMesh.GetMaxVertices_graphicsd3dxvb

Return Values

Returns the maximum number of vertices in the original mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSPMesh.GetNumFaces

#Retrieves the number of faces in the simplification mesh.

object.**GetNumFaces() As Long**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Return Values

Returns the number of faces in the simplification mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSPMesh.GetNumVertices

#Retrieves the number of vertices in the simplification mesh.

object.**GetNumVertices() As Long**

Parts

object

IDH_D3DXSPMesh.GetNumFaces_graphicsd3dxvb
IDH_D3DXSPMesh.GetNumVertices_graphicsd3dxvb

Object expression that resolves to a **D3DXSPMesh** object.

Return Values

Returns the number of vertices in the simplification mesh.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSPMesh.GetOptions

#Retrieves the mesh options enabled for this simplification mesh at creation time.

object.**GetOptions() As Long**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Return Values

Returns a combination of one or more of the flags defined by the **CONST_D3DXMESH** enumeration, indicating the options enabled for this mesh at creation time.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXSPMesh.ReduceFaces

#Reduces the number of faces in a simplification mesh.

object.**ReduceFaces(_**
Faces **As Long)**

IDH_D3DXSPMesh.GetOptions_graphicsd3dxvb
IDH_D3DXSPMesh.ReduceFaces_graphicsd3dxvb

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Faces
Number of faces to which to reduce the primitive count.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This method performs edge collapses to reduce the number of primitives to the value specified in *Faces*. It is not always possible to reduce a mesh to the requested value due to certain restrictions.

D3DXSPMesh.ReduceVertices

#Reduces the number of vertices in a simplification mesh.

object.**ReduceVertices(** _
Vertices **As Long)**

Parts

object
Object expression that resolves to a **D3DXSPMesh** object.

Vertices
Number of vertices to which to reduce the primitive count.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXSPMesh.ReduceVertices_graphicsd3dxvb

Remarks

This method performs edge collapses to reduce the number of primitives to the value specified in *Vertices*. It is not always possible to reduce a mesh to the requested value due to certain restrictions.

D3DXSprite

#The **D3DXSprite** class provides a set of methods that simplify the process of drawing sprites using Microsoft® Direct3D®.

The **D3DXSprite** class is obtained by calling the **D3DX8.CreateSprite** method.

The methods of the **D3DXSprite** class can be organized into the following groups.

Drawing	**Begin**
	Draw
	DrawTransform
	End
Information	**GetDevice**

D3DXSprite.Begin

#Prepares a device for drawing sprites.

object.**Begin()**

Parts

object
Object expression that resolves to a **D3DXSprite** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to one of the following values.

D3DERR_INVALIDCALL

D3DXERR_INVALIDDATA

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXSprite_graphicsd3dxvb
IDH_D3DXSprite.Begin_graphicsd3dxvb

Remarks

Calling **Begin** is optional. If called outside of a Begin/End sequence, the draw functions internally call **Begin** and **End**. To avoid extra overhead, this method should be used if more than one draw function is called successively.

D3DXSprite.Begin cannot be used as a substitute for either Direct3DDevice8.BeginScene or D3DXRenderToSurface.BeginScene.

See Also

D3DXSprite.End

D3DXSprite.Draw

#Draws a simple sprite in screen-space.

object.**Draw(** _
SrcTexture **As Direct3DTexture8**, _
SrcRect **As Any**, _
Scaling **As D3DVECTOR2**, _
RotationCenter **As D3DVECTOR2**, _
Rotation **As Single**, _
Translation **As D3DVECTOR2**, _
Color **As Long)**

Parts

object
Object expression that resolves to a **D3DXSprite** object.

SrcTexture
A **Direct3DTexture8** object, representing the source image used for the sprite.

SrcRect
A **RECT** type that indicates what portion of the source texture to use for the sprite. If this parameter is ByVal 0, then the entire source image is used for the sprite. However, you can specify a sub-rectangle of the source image instead. X and y mirroring can be specified easily by swapping the left and top and the right and bottom parameters of this **RECT** type.

Scaling
A **D3DVECTOR2** type, the scaling vector.

RotationCenter
A **D3DVECTOR2** type, a point that identifies the center of rotation.

Rotation
Value that specifies the rotation.

Translation

IDH_D3DXSprite.Draw_graphicsd3dxvb

D3DVECTOR2 type, representing the translation.

Color
A value describing the color. The color and alpha channels are modulated by this value.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If **D3DXSprite.Begin** has not been called, this method internally calls **Begin** and **D3DXSprite.End**. When making successive calls to **D3DXSprite.Draw** and/or **D3DXSprite.DrawTransform**, make sure to call **Begin** to avoid the extra overhead of **Draw** and **DrawTransform** internally calling **Begin** and **End** each time.

See Also

D3DXSprite.DrawTransform

D3DXSprite.DrawTransform

#Provides a mechanism for drawing a sprite that is transformed by a specified matrix.

object.**DrawTransform(** _
SrcTexture **As Direct3DTexture8**, _
SrcRect **As Any**, _
TransformMatrix **As D3DMATRIX**, _
Color **As Long)**

Parts

object
Object expression that resolves to a **D3DXSprite** object.

SrcTexture
Direct3DTexture8 object, representing the source image used for the sprite.

SrcRect
A **RECT** type that indicates what portion of the source texture to use for the sprite. If this parameter is ByVal 0, then the entire source image is used for the sprite. However, you can specify a sub-rectangle of the source image instead. X

IDH_D3DXSprite.DrawTransform_graphicsd3dxvb

and y mirroring can be specified easily by swapping the left and top and the right and bottom parameters of this **RECT** type.

Before transformation, the size of the sprite is defined by *SrcRect* with the top-left corner at the origin (0,0).

TransformMatrix
A **D3DMATRIX** type, specifying the transformation that is applied.

Color
Value representing the color. The color and alpha channels are modulated by this value.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If **D3DXSprite.Begin** has not been called, this method internally calls **Begin** and **D3DXSprite.End**. When making successive calls to **D3DXSprite.Draw** and/or **D3DXSprite.DrawTransform**, be sure to call **Begin** to avoid the extra overhead of **Draw** and **DrawTransform** internally calling **Begin** and **End** each time.

See Also

D3DXSprite.Draw

D3DXSprite.End

#Restores the device state to how it was when **D3DXSprite.Begin** was called.

object.**End()**

Parts

object
Object expression that resolves to a **D3DXSprite** object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

IDH_D3DXSprite.End_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

D3DXSprite.End cannot be used as a substitute for either **Direct3DDevice8.EndScene** or **D3DXRenderToSurface.EndScene**.

See Also

D3DXSprite.Begin

D3DXSprite.GetDevice

#Retrieves the Microsoft® Direct3D® device associated with the sprite object.

object.**GetDevice() As Direct3DDevice8**

Parts

object
Object expression that resolves to a **D3DXSprite** object.

Return Values

Direct3DDevice8 object representing the device object associated with the sprite object.

Error Codes

If the method fails, an error is raised and **Err.Number** can be set to D3DERR_INVALIDCALL.

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Functions

This section contains reference information for the functions that you need to implement when you work with Direct3DX. The following functions are implemented.

- Copy Functions
- Math Functions

IDH_D3DXSprite.GetDevice_graphicsd3dxvb

Copy Functions

#The copying library provided by the Microsoft® Direct3DX® utility library supplies functions that enable you to easily copy data between vertex arrays, index arrays, and vertex and index buffers associated with mesh objects.

Mesh Buffer	**D3DXMeshIndexBuffer8GetData**
	D3DXMeshIndexBuffer8SetData
	D3DXMeshVertexBuffer8GetData
	D3DXMeshVertexBuffer8SetData
Memory	**DXCopyMemory**

D3DXMeshIndexBuffer8GetData

#Copies indices from a vertex buffer attached to a mesh object to an index array.

D3DXMeshIndexBuffer8GetData(_
D3DXMeshobj **As Unknown**, _
Offset **As Long**, _
Size **As Long**, _
Flags **As Long**, _
Data **As Any) As Long**

Parameters

D3DXMeshobj
Object that resolves to a D3DXMesh object that contains the vertex buffer to retrieve.

Offset
Offset, in bytes, from the start of the buffer to where data retrieval begins.

Size
Size of the buffer, in bytes.

Flags
Combination of one or more valid locking flags defined by the CONST_D3DLOCKFLAGS enumeration, describing the type of lock to perform.

Data
Memory to be filled with the retrieved data. See Remarks.

IDH_Copy_Functions_graphicsd3dxvb
IDH_D3DXMeshIndexBuffer8GetData_graphicsd3dxvb

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following.

D3DERR_INVALIDCALL

E_INVALIDARG

Error Codes

Err.Number is not set for this function.

Remarks

The following code fragment demonstrates a typical use for this function.

```
Dim IndexArray(9) As Long
Dim Mesh As D3DXMesh
Dim hr As Long

'The following code assumes that Mesh has been properly initialized
hr = D3DXMeshIndexBuffer8GetData(Mesh, 0, (Len(IndexArray(0)) * 10), _
                                 0, IndexArray(0))
```

See Also

D3DXMeshIndexBuffer8SetData

D3DXMeshIndexBuffer8SetData

#Copies indices from an array of indices into an index buffer attached to a mesh object.

```
D3DXMeshIndexBuffer8SetData( _
    D3DXMeshobj As Unknown, _
    Offset As Long, _
    Size As Long, _
    Flags As Long, _
    Data As Any) As Long
```

Parameters

D3DXMeshobj

IDH_D3DXMeshIndexBuffer8SetData_graphicsd3dxvb

Object that resolves to a D3DXMesh object which contains the index buffer to set.

Offset
Offset, in bytes, from the start of the index buffer to where data is set.

Size
Size of the buffer, in bytes.

Flags
A combination of one or more valid locking flags defined by the CONST_D3DLOCKFLAGS enumeration, describing the type of lock to perform.

Data
The first element of an index array to be used to set the index buffer. See Remarks

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following.

D3DERR_INVALIDCALL

E_INVALIDARG

Error Codes

Err.Number is not set for this function.

Remarks

The following code fragment demonstrates a typical use for this function.

```
Dim IndexArray(9) As Long
Dim Mesh As D3DXMesh
Dim hr As Long

'The following code assumes that Mesh has been properly initialized
hr = D3DXMeshIndexBuffer8SetData(Mesh, 0, (Len(IndexArray(0)) * 10), _
                                 0, IndexArray(0))
```

See Also

D3DXMeshIndexBuffer8GetData

D3DXMeshVertexBuffer8GetData

#Copies vertices from a vertex buffer attached to a mesh object into an vertex array.

D3DXMeshVertexBuffer8GetData(_
D3DXMeshobj **As Unknown**, _
Offset **As Long**, _
Size **As Long**, _
Flags **As Long**, _
Data **As Any) As Long**

Parameters

D3DXMeshobj
Object that resolves to a D3DXMesh object that contains the vertex buffer to retrieve.

Offset
Offset, in bytes, from the start of the buffer to where data retrieval begins.

Size
Size of the buffer, in bytes.

Flags
A combination of one or more valid locking flags defined by the CONST_D3DLOCKFLAGS enumeration, describing the type of lock to perform.

Data
First element of an array of vertices to be filled with the retrieved data. See Remarks.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following.

D3DERR_INVALIDCALL

E_INVALIDARG

Error Codes

Err.Number is not set for this function.

Remarks

The following code fragment demonstrates a typical use for this function.

IDH_D3DXMeshVertexBuffer8GetData_graphicsd3dxvb

```
Dim VertArray(9) As D3DVERTEX
Dim Mesh As D3DXMesh
Dim hr As Long

'The following code assumes that Mesh has been properly initialized
hr = D3DXMeshVertexBuffer8GetData(Mesh, 0, (Len(VertArray(0)) * 10), _
                                  0, VertArray(0))
```

See Also

D3DXMeshVertexBuffer8SetData

D3DXMeshVertexBuffer8SetData

#Copies vertices from a vertex array into a vertex buffer attached to a mesh object.

D3DXMeshVertexBuffer8SetData(_
D3DXMeshobj **As Unknown**, _
Offset **As Long**, _
Size **As Long**, _
Flags **As Long**, _
Data **As Any) As Long**

Parameters

D3DXMeshobj
Object that resolves to a D3DXMesh object that contains the vertex buffer to set.

Offset
Offset, in bytes, from the start of the buffer to where data is set.

Size
Size of the buffer, in bytes.

Flags
A combination of one or more valid locking flags defined by the CONST_D3DLOCKFLAGS enumeration, describing the type of lock to perform.

Data
The first element of an array of vertices used to set the vertex buffer. See Remarks.

Return Values

If the function succeeds, the return value is D3D_OK.

If the function fails, the return value can be one of the following.

IDH_D3DXMeshVertexBuffer8SetData_graphicsd3dxvb

D3DERR_INVALIDCALL

E_INVALIDARG

Error Codes

Err.Number is not set for this function.

Remarks

The following code fragment demonstrates a typical use for this function.

```
Dim VertArray(9) As D3DVERTEX
Dim Mesh As D3DXMesh
Dim hr As Long

'The following code assumes that Mesh has been properly initialized
hr = D3DXMeshVertexBuffer8SetData(Mesh, 0, (Len(VertArray(0)) * 10), _
                                  0, VertArray(0))
```

See Also

D3DXMeshVertexBuffer8GetData

DXCopyMemory

#

DXCopyMemory(_
Dest **As Any**, _
Src **As Any**, _
Size **As Long) As Long**

Parameters

Dest
The memory into which to copy data. See Remarks.

Src
The memory from which to copy data. See Remarks

Size
Size, in bytes, of the data to copy.

Return Values

If the function succeeds, the return value is D3D_OK.

IDH_DXCopyMemory_graphicsd3dxvb

If the function fails, the return value can be one of the following.

D3DERR_INVALIDCALL

E_INVALIDARG

Error Codes

Err.Number is not set for this function.

Remarks

Both *Dest* and *Src* may be any of the following: a type, the first element of an array, a pointer returned by **D3DXBuffer.GetBufferPointer**, or a pointer returned from either **Direct3DIndexBuffer8.Lock** or **Direct3DVertexBuffer8.Lock**.

Math Functions

#The math library provided by the Direct3DX utility library supplies functions to compute both basic and complicated 3-D mathematical operations.

The 3-D math application functions can be organized into the following groups:

Color	**D3DXColorAdd**
	D3DXColorAdjustContrast
	D3DXColorAdjustSaturation
	D3DXColorLerp
	D3DXColorModulate
	D3DXColorNegative
	D3DXColorScale
	D3DXColorSubtract
Plane	**D3DXPlaneDot**
	D3DXPlaneDotCoord
	D3DXPlaneDotNormal
	D3DXPlaneFromPointNormal
	D3DXPlaneFromPoints
	D3DXPlaneIntersectLine
	D3DXPlaneNormalize

IDH_Math_Functions_graphicsd3dxvb

	D3DXPlaneTransform
Quaternion	**D3DXQuaternionBaryCentric**
	D3DXQuaternionConjugate
	D3DXQuaternionDot
	D3DXQuaternionExp
	D3DXQuaternionIdentity
	D3DXQuaternionInverse
	D3DXQuaternionIsIdentity
	D3DXQuaternionLength
	D3DXQuaternionLengthSq
	D3DXQuaternionLn
	D3DXQuaternionMultiply
	D3DXQuaternionNormalize
	D3DXQuaternionRotationAxis
	D3DXQuaternionRotationMatrix
	D3DXQuaternionRotationYawPitchRoll
	D3DXQuaternionSlerp
	D3DXQuaternionSquad
	D3DXQuaternionToAxisAngle
2-D Vector	**D3DXVec2Add**
	D3DXVec2BaryCentric
	D3DXVec2CatmullRom
	D3DXVec2CCW
	D3DXVec2Dot
	D3DXVec2Hermite
	D3DXVec2Length
	D3DXVec2LengthSq
	D3DXVec2Lerp
	D3DXVec2Maximize
	D3DXVec2Minimize

	D3DXVec2Normalize
	D3DXVec2Scale
	D3DXVec2Subtract
	D3DXVec2Transform
	D3DXVec2TransformCoord
	D3DXVec2TransformNormal
3-D Vector	**D3DXVec3Add**
	D3DXVec3BaryCentric
	D3DXVec3CatmullRom
	D3DXVec3Cross
	D3DXVec3Dot
	D3DXVec3Hermite
	D3DXVec3Length
	D3DXVec3LengthSq
	D3DXVec3Lerp
	D3DXVec3Maximize
	D3DXVec3Minimize
	D3DXVec3Normalize
	D3DXVec3Project
	D3DXVec3Scale
	D3DXVec3Subtract
	D3DXVec3Transform
	D3DXVec3TransformCoord
	D3DXVec3TransformNormal
	D3DXVec3Unproject
4-D Matrix	**D3DXMatrixAffineTransformation**
	D3DXMatrixfDeterminant
	D3DXMatrixIdentity
	D3DXMatrixInverse
	D3DXMatrixIsIdentity
	D3DXMatrixLookAtLH

	D3DXMatrixLookAtRH
	D3DXMatrixMultiply
	D3DXMatrixOrthoLH
	D3DXMatrixOrthoRH
	D3DXMatrixOrthoOffCenterLH
	D3DXMatrixOrthoOffCenterRH
	D3DXMatrixPerspectiveFovLH
	D3DXMatrixPerspectiveFovRH
	D3DXMatrixPerspectiveLH
	D3DXMatrixPerspectiveRH
	D3DXMatrixPerspectiveOffCenterLH
	D3DXMatrixPerspectiveOffCenterRH
	D3DXMatrixReflect
	D3DXMatrixRotationAxis
	D3DXMatrixRotationQuaternion
	D3DXMatrixRotationX
	D3DXMatrixRotationY
	D3DXMatrixRotationYawPitchRoll
	D3DXMatrixRotationZ
	D3DXMatrixScaling
	D3DXMatrixShadow
	D3DXMatrixTransformation
	D3DXMatrixTranslation
	D3DXMatrixTranspose
4-D Vector	**D3DXVec4Add**
	D3DXVec4BaryCentric
	D3DXVec4CatmullRom
	D3DXVec4Cross
	D3DXVec4Dot
	D3DXVec4Hermite
	D3DXVec4Length

D3DXVec4LengthSq

D3DXVec4Lerp

D3DXVec4Maximize

D3DXVec4Minimize

D3DXVec4Normalize

D3DXVec4Scale

D3DXVec4Subtract

D3DXVec4Transform

D3DXColorAdd

#Adds two color values together to create a new color value.

D3DXColorAdd(_
COut **As D3DCOLORVALUE**, _
C1 **As D3DCOLORVALUE**, _
C2 **As D3DCOLORVALUE)**

Parameters

COut
A **D3DCOLORVALUE** type that is the result of the operation, the sum of two color values.

C1
A source **D3DCOLORVALUE** type.

C2
A source **D3DCOLORVALUE** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXColorAdd_graphicsd3dxvb

See Also

D3DXColorModulate, **D3DXColorSubtract**

D3DXColorAdjustContrast

#Adjusts the contrast value of a color.

```
D3DXColorAdjustContrast( _
    COut As D3DCOLORVALUE, _
    C1 As D3DCOLORVALUE, _
    c As Single)
```

Parameters

COut

A **D3DCOLORVALUE** type that is the result of the operation, the result of the contrast adjustment.

C1

A source **D3DCOLORVALUE** type.

c

Contrast value. This parameter linearly interpolates between 50 percent gray and the color *C1*. There are not limits on the value of *c*. If this parameter is zero, then the returned color is 50 percent gray. If this parameter is 1, then the returned color is the original color.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function interpolates the red, green, and blue color components of a **D3DCOLORVALUE** type between 50 percent gray and a specified contrast value, as shown in the following example.

```
Cout.r = 0.5 + c * (C1.r - 0.5)
```

IDH_D3DXColorAdjustContrast_graphicsd3dxvb

If *c* is greater than 0 and less than 1, the contrast is decreased. If *c* is greater than 1, then the contrast is increased.

See Also

D3DXColorAdjustSaturation

D3DXColorAdjustSaturation

#Adjusts the saturation value of a color.

```
D3DXColorAdjustSaturation( _
    COut As D3DCOLORVALUE, _
    C1 As D3DCOLORVALUE, _
    s As Single)
```

Parameters

COut

A **D3DCOLORVALUE** type that is the result of the operation, the result of the saturation adjustment.

C1

A source **D3DCOLORVALUE** type.

s

Saturation value. This parameter linearly interpolates between the color converted to gray-scale and the original color, *C1*. There are no limits on the value of *s*. If *s* is 0, then the returned color is the gray-scale color. If *s* is 1, the returned color is the original color.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function interpolates the red, green, and blue color components of a **D3DCOLORVALUE** type between an unsaturated color and a color, as shown in the following example.

IDH_D3DXColorAdjustSaturation_graphicsd3dxvb

```
' Approximate values for each component's contribution to luminance.
' Based up the NTSC standard described in ITU-R Recommendation BT.709.
Dim Grey As Long
Grey = C1.r * 0.2125 + C1.g * 0.7154 + C1.b * 0.0721

COut.r = Grey + s * (C1.r - Grey)
```

If *s* is greater than 0 and less than 1, the saturation is decreased. If *s* is greater than 1, the saturation is increased.

The gray-scale color is computed as: r = g = b = 0.2125*r + 0.7154*g + 0.0721*b.

See Also

D3DXColorAdjustContrast

D3DXColorLerp

#Uses linear interpolation to create a color value.

D3DXColorLerp(_
COut **As D3DCOLORVALUE**, _
C1 **As D3DCOLORVALUE**, _
C2 **As D3DCOLORVALUE**, _
s **As Single)**

Parameters

COut
A **D3DCOLORVALUE** type that is the result of the operation, the result of the linear interpolation.

C1
A source **D3DCOLORVALUE** type.

C2
A source **D3DCOLORVALUE** type.

s
Parameter that linearly interpolates between the colors, *C1* and *C2*, treating them both as 4-D vectors. There are no limits on the value of *s*.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

IDH_D3DXColorLerp_graphicsd3dxvb

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function interpolates the red, green, blue, and alpha components of a **D3DCOLORVALUE** type between two colors, as shown in the following example.

```
COut.r = .C1.r + s * (C2.r - C1.r)
```

If you are linearly interpolating between the colors A and B, and *s* is 0, the resulting color is A. If *s* is 1, the resulting color is B.

See Also

D3DXColorModulate, **D3DXColorNegative**, **D3DXColorScale**

D3DXColorModulate

#Blends two colors.

D3DXColorModulate(_
COut **As D3DCOLORVALUE**, _
C1 **As D3DCOLORVALUE**, _
C2 **As D3DCOLORVALUE)**

Parameters

COut
A **D3DCOLORVALUE** type that is the result of the operation, the result of the blending operation.

C1
A source **D3DCOLORVALUE** type.

C2
A source **D3DCOLORVALUE** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

IDH_D3DXColorModulate_graphicsd3dxvb

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function blends together two colors by multiplying matching color components, as shown in the following example.

```
Cout.r = C1.r * C2.r
```

See Also

D3DXColorLerp, **D3DXColorNegative**, **D3DXColorScale**

D3DXColorNegative

#Creates the negative color value of a color value.

D3DXColorNegative(_
COut **As D3DCOLORVALUE**, **_**
C **As D3DCOLORVALUE)**

Parameters

COut
A **D3DCOLORVALUE** type that is the result of the operation, the negative color value of the color value.

C
A source **D3DCOLORVALUE** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXColorNegative_graphicsd3dxvb

Remarks

This function returns the negative color value by subtracting 1.0 from the color components of the **D3DCOLORVALUE** type, as shown in the following example.

```
COut.r = 1.0 - C.r
```

See Also

D3DXColorLerp, **D3DXColorModulate**, **D3DXColorScale**

D3DXColorScale

#Scales a color value.

D3DXColorScale(_
COut **As D3DCOLORVALUE, _**
C1 **As D3DCOLORVALUE, _**
s **As Single)**

Parameters

COut
A **D3DCOLORVALUE** type that is the result of the operation, the scaled color value.

C1
A source **D3DCOLORVALUE** type.

s
Scale factor. It scales the color, treating it like a 4-D vector. There are no limits on the value of *s*. If *s* is 1, the resulting color is the original color.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXColorScale_graphicsd3dxvb

Remarks

This function computes the scaled color value by multiplying the color components of the **D3DCOLORVALUE** type by the specified scale factor, as shown in the following example.

```
Cout.r = C1.r * s
```

See Also

D3DXColorLerp, **D3DXColorModulate**, **D3DXColorNegative**

D3DXColorSubtract

#Subtracts two color values to create a new color value.

D3DXColorSubtract(_
COut **As D3DCOLORVALUE**, **_**
C1 **As D3DCOLORVALUE**, **_**
C2 **As D3DCOLORVALUE)**

Parameters

COut
A **D3DCOLORVALUE** type that is the result of the operation, the difference between two color values.

C1
A source **D3DCOLORVALUE** type.

C2
A source **D3DCOLORVALUE** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXColorSubtract_graphicsd3dxvb

See Also

D3DXColorAdd

D3DXMatrixAffineTransformation

#Builds an affine transformation matrix.

```
D3DXMatrixAffineTransformation( _
    MOut As D3DMATRIX, _
    Scaling As Single, _
    VRotationCenter As D3DVECTOR, _
    QRotation As D3DQUATERNION, _
    VTranslation As D3DVECTOR)
```

Parameters

MOut
D3DMATRIX type that is the result of the operation, an affine transformation matrix.

Scaling
Scaling factor.

VRotationCenter
D3DVECTOR type, a point identifying the center of rotation. If this argument is an empty **D3DVECTOR** type, it is treated as identity.

QRotation
D3DQUATERNION type that specifies the rotation. If this argument is an empty **D3DQUATERNION** type, it is treated as identity.

VTranslation
D3DVECTOR type, representing the translation. If this argument is an empty **D3DVECTOR** type, it is treated as identity.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXMatrixAffineTransformation_graphicsd3dxvb

Remarks

The **D3DXMatrixAffineTransformation** function calculates the affine transformation matrix with the following formula: Ms * Mrc-1 * Mr * Mrc * Mt, where Ms is the scaling matrix, Mrc is the center of rotation matrix, Mr is the rotation matrix, and Mt is the translation matrix.

D3DXMatrixfDeterminant

#Returns the determinant of a matrix.

D3DXMatrixfDeterminant(_
M **As D3DMATRIX) As Single**

Parameters

M
The source **D3DMATRIX** type.

Return Values

The value of the matrix, the determinant.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXMatrixIdentity

#Creates an identity matrix.

D3DXMatrixIdentity(_
MOut **As D3DMATRIX)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, the identity matrix.

IDH_D3DXMatrixfDeterminant_graphicsd3dxvb
IDH_D3DXMatrixIdentity_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The identity matrix is a matrix in which all coefficients are 0 except the [1,1][2,2] [3,3][4,4] coefficients, which are set to 1. The identity matrix is special in that when it is applied to vertices, they are unchanged. The identity matrix is used as the starting point for matrices that will modify vertex values to create rotations, translations, and any other transformations that can be represented by a 4×4 matrix.

See Also

D3DXMatrixIsIdentity

D3DXMatrixInverse

[#]Calculates the inverse of a matrix.

D3DXMatrixInverse(_
MOut **As D3DMATRIX**, _
Determinant **As Single**, _
M **As D3DMATRIX)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, the inverse of the matrix.

Determinant
After this function returns, a **Single** value that is the determinant of the matrix, *M*.

M
The source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

[#] IDH_D3DXMatrixInverse_graphicsd3dxvb

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If matrix inversion fails, an empty **D3DMATRIX** type is returned by the **D3DXMatrixInverse** function. The determinant of *M* is returned if the *Determinant* parameter is not 0.

D3DXMatrixIsIdentity

#Determines if a matrix is an identity matrix.

D3DXMatrixIsIdentity(_
M **As D3DMATRIX) As Long**

Parameters

M
D3DMATRIX type that will be tested for identity.

Return Values

If the matrix is an identity matrix, this function returns a non-zero value (True); otherwise, this function returns zero (False).

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixIdentity

IDH_D3DXMatrixIsIdentity_graphicsd3dxvb

D3DXMatrixLookAtRH

#Builds a right-handed, look-at matrix.

D3DXMatrixLookAtRH(_
MOut **As D3DMATRIX**, _
VEye **As D3DVECTOR**, _
VAt **As D3DVECTOR**, _
VUp **As D3DVECTOR)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, a right-handed, look-at matrix.

VEye
D3DVECTOR type that defines the eye point. This value is used in translation.

VAt
D3DVECTOR type that defines the camera look-at target.

VUp
D3DVECTOR type that defines the current world's up, usually [0, 1, 0].

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixLookAtLH

D3DXMatrixLookAtLH

#Builds a left-handed, look-at matrix.

D3DXMatrixLookAtLH(_
MOut **As D3DMATRIX**, _
VEye **As D3DVECTOR**, _

IDH_D3DXMatrixLookAtRH_graphicsd3dxvb
IDH_D3DXMatrixLookAtLH_graphicsd3dxvb

VAt **As D3DVECTOR**, _
VUp **As D3DVECTOR)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, a left-handed, look-at matrix.

VEye
D3DVECTOR type that defines the eye point. This value is used in translation.

VAt
D3DVECTOR type that defines the camera look-at target.

VUp
D3DVECTOR type that defines the current world's up, usually [0, 1, 0].

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixLookAtRH

D3DXMatrixMultiply

#Determines the product of two matrices.

D3DXMatrixMultiply(_
MOut **As D3DMATRIX**, _
M1 **As D3DMATRIX**, _
M2 **As D3DMATRIX)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, the product of two matrices.

IDH_D3DXMatrixMultiply_graphicsd3dxvb

M1
A source **D3DMATRIX** type.

M2
A source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The result represents the transformation M2 followed by the transformation M1 (Out = M1 * M2).

See Also

D3DXQuaternionMultiply

D3DXMatrixOrthoRH

#Builds a right-handed orthogonal projection matrix.

```
D3DXMatrixOrthoRH( _
    MOut As D3DMATRIX, _
    w As Single, _
    h As Single, _
    zn As Single, _
    zf As Single)
```

Parameters

MOut
D3DMATRIX type that is the result of the operation, a right-handed orthogonal projection matrix.

w
Width of the view-volume.

h
Height of the view-volume.

IDH_D3DXMatrixOrthoRH_graphicsd3dxvb

zn
Minimum z-value of the view volume.

zf
Maximum z-value of the view volume.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

An orthogonal matrix is an invertible matrix for which the inverse of the matrix is equal to the transpose of the matrix.

All the parameters of the **D3DXMatrixOrtho** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

See Also

D3DXMatrixOrthoLH, **D3DXMatrixOrthoOffCenterRH**, **D3DXMatrixOrthoOffCenterLH**

D3DXMatrixOrthoLH

#Builds a left-handed orthogonal projection matrix.

```
D3DXMatrixOrthoLH( _
    MOut As D3DMATRIX, _
    w As Single, _
    h As Single, _
    zn As Single, _
    zf As Single)
```

Parameters

MOut
D3DMATRIX type that is the result of the operation, a left-handed orthogonal projection matrix.

w

IDH_D3DXMatrixOrthoLH_graphicsd3dxvb

Width of the view-volume.

h

Height of the view-volume.

zn

Minimum z-value of the view volume.

zf

Maximum z-value of the view volume.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

An orthogonal matrix is an invertible matrix for which the inverse of the matrix is equal to the transpose of the matrix.

All the parameters of the **D3DXMatrixOrthoLH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

See Also

D3DXMatrixOrthoRH, **D3DXMatrixOrthoOffCenterRH**, **D3DXMatrixOrthoOffCenterLH**

D3DXMatrixOrthoOffCenterRH

#Builds a customized, right-handed orthogonal projection matrix.

```
D3DXMatrixOrthoOffCenterRH( _
    MOut As D3DMATRIX, _
    l As Single, _
    r As Single, _
    b As Single, _
    t As Single, _
    zn As Single, _
    zf As Single)
```

IDH_D3DXMatrixOrthoOffCenterRH_graphicsd3dxvb

Parameters

MOut
D3DMATRIX type that is the result of the operation, a customized, right-handed orthogonal projection matrix.

l
Minimum x-value of view-volume.

r
Maximum x-value of view-volume.

b
Maximum y-value of view-volume.

t
Minimum y-value of view-volume.

zn
Minimum z-value of the view volume.

zf
Maximum z-value of the view volume.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

An orthogonal matrix is an invertible matrix for which the inverse of the matrix is equal to the transpose of the matrix.

The **D3DXMatrixOrthoRH** function is a special case of the **D3DXMatrixOrthoOffCenterRH** function. To create the same projection using **D3DXMatrixOrthoOffCenterRH**, use the following values: l = -w/2, r = w/2, b = -h/2, and t = h/2.

All the parameters of the **D3DXMatrixOrthoOffCenterRH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

See Also

D3DXMatrixOrthoRH, **D3DXMatrixOrthoLH**, **D3DXMatrixOrthoOffCenterLH**

D3DXMatrixOrthoOffCenterLH

#Builds a customized, left-handed orthogonal projection matrix.

D3DXMatrixOrthoOffCenterLH(_
MOut **As D3DMATRIX**, _
l **As Single**, _
r **As Single**, _
b **As Single**, _
t **As Single**, _
zn **As Single**, _
zf **As Single)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, a customized, left-handed orthogonal projection matrix.

l
Minimum x-value of view-volume.

r
Maximum x-value of view-volume.

b
Maximum y-value of view-volume.

t
Minimum y-value of view-volume.

zn
Minimum z-value of the view volume.

zf
Maximum z-value of the view volume.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXMatrixOrthoOffCenterLH_graphicsd3dxvb

Remarks

An orthogonal matrix is an invertible matrix for which the inverse of the matrix is equal to the transpose of the matrix.

The **D3DXMatrixOrthoLH** function is a special case of the **D3DXMatrixOrthoOffCenterLH** function. To create the same projection using **D3DXMatrixOrthoOffCenterLH**, use the following values: *l* = -w/2, *r* = w/2, *b* = -h/2, and *t* = h/2.

All the parameters of the **D3DXMatrixOrthoOffCenterLH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

See Also

D3DXMatrixOrthoRH, **D3DXMatrixOrthoLH**, **D3DXMatrixOrthoOffCenterRH**

D3DXMatrixPerspectiveRH

#Builds a right-handed perspective projection matrix.

```
D3DXMatrixPerspectiveRH( _
    MOut As D3DMATRIX, _
    w As Single, _
    h As Single, _
    zn As Single, _
    zf As Single)
```

Parameters

MOut
D3DMATRIX type that is the result of the operation, a right-handed perspective projection matrix.

w
Width of the view-volume at the near view-plane.

h
Height of the view-volume at the near view-plane.

zn
Z-value of the near view-plane.

zf
Z-value of the far view-plane.

IDH_D3DXMatrixPerspectiveRH_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

All the parameters of the **D3DXMatrixPerspectiveRH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

See Also

D3DXMatrixPerspectiveLH, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterRH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveFovLH

#Builds a left-handed perspective projection matrix based on a field of view (FOV).

```
D3DXMatrixPerspectiveFovLH( _
    MOut As D3DMATRIX, _
    fovy As Single, _
    aspect As Single, _
    zn As Single, _
    zf As Single)
```

Parameters

MOut
D3DMATRIX type that is the result of the operation, left-handed perspective projection matrix.

fovy
Field of view, in radians.

aspect
Aspect ratio.

zn
Z-value of the near view-plane.

IDH_D3DXMatrixPerspectiveFovLH_graphicsd3dxvb

zf
Z-value of the far view-plane.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveLH**, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveOffCenterRH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveFovRH

#Builds a right-handed perspective projection matrix based on a field of view (FOV).

D3DXMatrixPerspectiveFovRH(_
MOut **As D3DMATRIX**, _
fovy **As Single**, _
aspect **As Single**, _
zn **As Single**, _
zf **As Single)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, right-handed perspective projection matrix.

fovy
Field of view, in radians.

aspect
Aspect ratio.

zn
Z-value of the near view-plane.

zf
Z-value of the far view-plane.

IDH_D3DXMatrixPerspectiveFovRH_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveLH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterRH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveLH

#Builds a left-handed perspective projection matrix

D3DXMatrixPerspectiveLH(_
MOut **As D3DMATRIX**, _
w **As Single**, _
h **As Single**, _
zn **As Single**, _
zf **As Single)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, a left-handed perspective projection matrix.

w
Width of the view-volume at the near view-plane.

h
Height of the view-volume at the near view-plane.

zn
Z-value of the near view-plane.

zf
Z-value of the far view-plane.

IDH_D3DXMatrixPerspectiveLH_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

All the parameters of the **D3DXMatrixPerspectiveLH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterRH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveOffCenterRH

#Builds a customized, right-handed perspective projection matrix.

```
D3DXMatrixPerspectiveOffCenterRH( _
    MOut As D3DMATRIX, _
    l As Single, _
    r As Single, _
    b As Single, _
    t As Single, _
    zn As Single, _
    zf As Single)
```

Parameters

MOut

D3DMATRIX type that is the result of the operation, a customized, right-handed perspective projection matrix.

l

X-value of the near view-plane.

r

X-value of the far view-plane.

b

IDH_D3DXMatrixPerspectiveOffCenterRH_graphicsd3dxvb

Y-value of the far view-plane.

t
Y-value of the near view-plane.

zn
Z-value of the near view-plane.

zf
Z-value of the far view-plane.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

All the parameters of the **D3DXMatrixPerspectiveOffCenterRH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveLH**, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterLH**

D3DXMatrixPerspectiveOffCenterLH

#Builds a customized, left-handed perspective projection matrix.

```
D3DXMatrixPerspectiveOffCenterLH( _
    MOut As D3DMATRIX, _
    l As Single, _
    r As Single, _
    b As Single, _
    t As Single, _
    zn As Single, _
    zf As Single)
```

IDH_D3DXMatrixPerspectiveOffCenterLH_graphicsd3dxvb

Parameters

MOut
D3DMATRIX type that is the result of the operation, a customized, left-handed perspective projection matrix.

l
X-value of the near view-plane.

r
X-value of the far view-plane.

b
Y-value of the far view-plane.

t
Y-value of the near view-plane.

zn
Z-value of the near view-plane.

zf
Z-value of the far view-plane.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

All the parameters of the **D3DXMatrixPerspectiveOffCenterLH** function are distances in camera-space. The parameters describe the dimensions of the view-volume.

See Also

D3DXMatrixPerspectiveRH, **D3DXMatrixPerspectiveLH**, **D3DXMatrixPerspectiveFovRH**, **D3DXMatrixPerspectiveFovLH**, **D3DXMatrixPerspectiveOffCenterRH**

D3DXMatrixReflect

#Builds a matrix that reflects the coordinate system about a plane.

IDH_D3DXMatrixReflect_graphicsd3dxvb

D3DXMatrixReflect(_
MOut **As D3DMATRIX**, _
Plane **As D3DPLANE)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, a matrix that reflects the coordinate system about the source plane.

Plane
The source **D3DPLANE** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXMatrixRotationAxis

#Builds a matrix that rotates around an arbitrary axis.

D3DXMatrixRotationAxis(_
MOut **As D3DMATRIX**, _
VAxis **As D3DVECTOR**, _
angle **As Single)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, a matrix rotated around the specified axis.

VAxis
D3DVECTOR type that identifies the axis angle.

angle
Angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

IDH_D3DXMatrixRotationAxis_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixRotationQuaternion, **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, **D3DXMatrixRotationYawPitchRoll**, **D3DXMatrixRotationZ**

D3DXMatrixRotationQuaternion

#Builds a matrix from a quaternion.

D3DXMatrixRotationQuaternion(_
MOut **As D3DMATRIX**, _
Q **As D3DQUATERNION)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, a matrix built from the source quaternion.

Q
The source **D3DQUATERNION** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXMatrixRotationQuaternion_graphicsd3dxvb

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, **D3DXMatrixRotationYawPitchRoll**, **D3DXMatrixRotationZ**

D3DXMatrixRotationX

#Builds a matrix that rotates around the x-axis.

D3DXMatrixRotationX(_
MOut **As D3DMATRIX**, _
angle **As Single)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, a matrix rotated around the x-axis.

angle
Angle of rotation in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationQuaternion**, **D3DXMatrixRotationY**, **D3DXMatrixRotationYawPitchRoll**, **D3DXMatrixRotationZ**

D3DXMatrixRotationY

#Builds a matrix that rotates around the y-axis.

D3DXMatrixRotationY(_
MOut **As D3DMATRIX**, _

IDH_D3DXMatrixRotationX_graphicsd3dxvb
IDH_D3DXMatrixRotationY_graphicsd3dxvb

angle **As Single)**

Parameters

MOut

D3DMATRIX type that is the result of the operation, a matrix rotated around the y-axis.

angle

Angle of rotation in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationQuaternion**, **D3DXMatrixRotationX**, **D3DXMatrixRotationYawPitchRoll**, **D3DXMatrixRotationZ**

D3DXMatrixRotationYawPitchRoll

#Builds a matrix with a specified yaw, pitch, and roll.

D3DXMatrixRotationYawPitchRoll(_
MOut **As D3DMATRIX**, _
yaw **As Single**, _
pitch **As Single**, _
roll **As Single)**

Parameters

MOut

D3DMATRIX type that is the result of the operation, a matrix with the specified yaw, pitch, and roll.

yaw

Yaw around the y-axis, in radians.

IDH_D3DXMatrixRotationYawPitchRoll_graphicsd3dxvb

pitch
Pitch around the x-axis, in radians.

roll
Roll around the z-axis, in radians.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationQuaternion**, **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, **D3DXMatrixRotationZ**

D3DXMatrixRotationZ

#Builds a matrix that rotates around the z-axis.

```
D3DXMatrixRotationZ( _
    MOut As D3DMATRIX, _
    angle As Single)
```

Parameters

MOut
D3DMATRIX type that is the result of the operation, a matrix rotated around the z-axis.

angle
Angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXMatrixRotationZ_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXMatrixRotationAxis, **D3DXMatrixRotationQuaternion**, **D3DXMatrixRotationX**, **D3DXMatrixRotationY**, **D3DXMatrixRotationYawPitchRoll**

D3DXMatrixScaling

#Builds a matrix that scales along the x-, y-, and z-axes.

```
D3DXMatrixScaling( _
    MOut As D3DMATRIX, _
    x As Single, _
    y As Single, _
    z As Single)
```

Parameters

MOut
D3DMATRIX type that is the result of the operation, the scaled matrix.

x
Scaling factor that is applied along the x-axis.

y
Scaling factor that is applied along the y-axis.

z
Scaling factor that is applied along the z-axis.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXMatrixScaling_graphicsd3dxvb

D3DXMatrixShadow

#Builds a matrix that projects geometry into a plane, as if casting a shadow from a light. See Remarks.

D3DXMatrixShadow(_
MOut **As D3DMATRIX**, _
VLight **As D3DVECTOR4**, _
Plane **As D3DPLANE)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, a matrix that projects geometry into a plane. See Remarks.

VLight
D3DVECTOR4 type describing the light's position.

Plane
The source **D3DPLANE** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXMatrixShadow** function builds a matrix that projects geometry into a plane. If *Vlight*.w is 0, then the ray from the origin to *Vlight* represents a directional light. If it is 1, then *Vlight* represents a point light.

D3DXMatrixTransformation

#Builds a transformation matrix.

D3DXMatrixTransformation(_
MOut **As D3DMATRIX**, _
VScaleCenter **As D3DVECTOR**, _

IDH_D3DXMatrixShadow_graphicsd3dxvb
IDH_D3DXMatrixTransformation_graphicsd3dxvb

QScaleRotation **As D3DQUATERNION**, _
VScale **As D3DVECTOR**, _
VRotationCenter **As D3DVECTOR**, _
QRotation **As D3DQUATERNION**, _
VTranslation **As D3DVECTOR)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, the transformation matrix.

VScaleCenter
D3DVECTOR type, identifying the scaling center point. If this argument is an empty **D3DVECTOR** type, it is treated as identity.

QScaleRotation
D3DQUATERNION type that specifies the scaling rotation If this argument is an empty **D3DQUATERNION** type, it is treated as identity.

VScale
D3DVECTOR type, the scaling vector. If this argument is an empty **D3DVECTOR** type, it is treated as identity.

VRotationCenter
D3DVECTOR type, a point that identifies the center of rotation. If this argument is an empty **D3DVECTOR** type, it is treated as identity.

QRotation
D3DQUATERNION type that specifies the rotation. If this argument is an empty **D3DQUATERNION** type, it is treated as identity.

VTranslation
D3DVECTOR type, representing the translation. If this argument is an empty **D3DVECTOR** type, it is treated as identity.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXMatrixTransformation** function calculates the transformation matrix with the following formula: Msc-1 * Msr-1 * Ms * Msr * Msc * Mrc-1 * Mr * Mrc * Mt, where Msc is the center scaling matrix, Msr is the scaling rotation matrix, Ms is

the scaling matrix, Mrc is the center of rotation matrix, Mr is the rotation matrix, and Mt is the translation matrix.

See Also

D3DXMatrixAffineTransformation

D3DXMatrixTranslation

#Builds a matrix using the specified offsets.

```
D3DXMatrixTranslation( _
   MOut As D3DMATRIX, _
   x As Single, _
   y As Single, _
   z As Single)
```

Parameters

MOut
D3DMATRIX type that is the result of the operation, the translated matrix.

x
X-coordinate offset.

y
Y-coordinate offset.

z
Z-coordinate offset.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXMatrixTranspose

#Returns the matrix transpose of a matrix.

```
D3DXMatrixTranspose( _
```

IDH_D3DXMatrixTranslation_graphicsd3dxvb
IDH_D3DXMatrixTranspose_graphicsd3dxvb

MOut **As D3DMATRIX**, _
M **As D3DMATRIX)**

Parameters

MOut
D3DMATRIX type that is the result of the operation, the matrix transpose of the matrix.

M
The source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXPlaneDot

#Computes the dot-product of a plane and a 4-D vector.

D3DXPlaneDot(_
P1 **As D3DPLANE**, _
P2 **As D3DPLANE) As Single**

Parameters

P1
A source **D3DPLANE** type.

P2
A source **D3DPLANE** type.

Return Values

The dot-product of the plane and 4-D vector.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

IDH_D3DXPlaneDot_graphicsd3dxvb

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Given a plane (a, b, c, d) and a 4-D vector (x, y, z, w) the return value of this function is a*x + b*y + c*z + d*w. The **D3DXPlaneDot** function is useful for determining the plane's relationship with a homogeneous coordinate. For example, this function could be used to determine if a particular coordinate is on a particular plane, or on which side of a particular plane a particular coordinate lies.

See Also

D3DXPlaneDotCoord, **D3DXPlaneDotNormal**

D3DXPlaneDotCoord

#Computes the dot-product of a plane and a 3-D vector. The *w* parameter of the vector is assumed to be 1.

```
D3DXPlaneDotCoord( _
   P1 As D3DPLANE, _
   V As D3DVECTOR) As Single
```

Parameters

P1

A source **D3DPLANE** type.

V

A source **D3DVECTOR** type.

Return Values

The dot-product of the plane and 3-D vector.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXPlaneDotCoord_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Given a plane (a, b, c, d) and a 3-D vector (x, y, z) the return value of this function is a*x + b*y + c*z + d*1. The **D3DXPlaneDotCoord** function is useful for determining the plane's relationship with a coordinate in 3-D space.

See Also

D3DXPlaneDot, **D3DXPlaneDotNormal**

D3DXPlaneDotNormal

#Computes the dot-product of a plane and a 3-D vector. The *w* parameter of the vector is assumed to be 0.

```
D3DXPlaneDotNormal( _
   P1 As D3DPLANE, _
   V As D3DVECTOR) As Single
```

Parameters

P1

A source **D3DPLANE** type.

V

A source **D3DVECTOR** type.

Return Values

The dot-product of the plane and 3-D vector.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXPlaneDotNormal_graphicsd3dxvb

Remarks

Given a plane (a, b, c, d) and a 3-D vector (x, y, z) the return value of this function is a*x + b*y + c*z + d*0. The **D3DXPlaneDotNormal** function is useful for figuring out the angle between the normal of the plane, and another normal.

See Also

D3DXPlaneDot, **D3DXPlaneDotCoord**

D3DXPlaneFromPointNormal

#Constructs a plane from a point and a normal.

```
D3DXPlaneFromPointNormal( _
    POut As D3DPLANE, _
    VPoint As D3DVECTOR, _
    vNormal As D3DVECTOR)
```

Parameters

POut
D3DPLANE type that is the result of the operation, a plane constructed from the point and the normal.

VPoint
D3DVECTOR type, defining the point used to construct the plane.

vNormal
D3DVECTOR type, defining the normal used to construct the plane.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXPlaneFromPoints

IDH_D3DXPlaneFromPointNormal_graphicsd3dxvb

D3DXPlaneFromPoints

#Constructs a plane from three points.

D3DXPlaneFromPoints(_
POut **As D3DPLANE**, _
V1 **As D3DVECTOR**, _
V2 **As D3DVECTOR**, _
V3 **As D3DVECTOR)**

Parameters

POut
D3DPLANE type that is the result of the operation, a plane constructed from the given points.

V1
D3DVECTOR type, defining one of the points used to construct the plane.

V2
D3DVECTOR type, defining one of the points used to construct the plane.

V3
D3DVECTOR type, defining one of the points used to construct the plane.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXPlaneFromPointNormal

D3DXPlaneIntersectLine

#Finds the intersection between a plane and a line.

D3DXPlaneIntersectLine(_
POut **As D3DPLANE**, _
P **As D3DPLANE**, _

IDH_D3DXPlaneFromPoints_graphicsd3dxvb
IDH_D3DXPlaneIntersectLine_graphicsd3dxvb

V1 **As D3DVECTOR**, _
V2 **As D3DVECTOR)**

Parameters

POut
D3DPLANE type that is the result of the operation, identifying the intersection between the specified plane and line.

P
The source **D3DPLANE** type.

V1
A source **D3DVECTOR** type, defining a line starting point.

V2
A source **D3DVECTOR** type, defining a line ending point.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

If the line is parallel to the plane, an empty **D3DPLANE** type is returned.

D3DXPlaneNormalize

#Returns the normal of a plane.

D3DXPlaneNormalize(_
POut **As D3DPLANE**, _
P **As D3DPLANE)**

Parameters

POut
D3DPLANE type that is the result of the operation, the normal of the plane.

P
The source **D3DPLANE** type.

IDH_D3DXPlaneNormalize_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

D3DXPlaneNormalize normalizes a plane so that |a,b,c| == 1.

D3DXPlaneTransform

#Transforms a plane by a given matrix.

D3DXPlaneTransform(_
POut **As D3DPLANE**, _
P **As D3DPLANE**, _
M **As D3DMATRIX)**

Parameters

POut
D3DPLANE type that is the result of the operation, the transformed plane.

P
The source **D3DPLANE** type.

M
The source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXPlaneTransform_graphicsd3dxvb

Remarks

The matrix, *M*, must be an affine transform.

D3DXQuaternionBaryCentric

#Returns a quaternion in Barycentric coordinates.

```
D3DXQuaternionBaryCentric( _
    QOut As D3DQUATERNION, _
    Q1 As D3DQUATERNION, _
    Q2 As D3DQUATERNION, _
    Q3 As D3DQUATERNION, _
    f As Single, _
    g As Single)
```

Parameters

QOut
D3DQUATERNION type that is the result of the operation, a quaternion in Barycentric coordinates.

Q1
A source **D3DQUATERNION** type.

Q2
A source **D3DQUATERNION** type.

Q3
A source **D3DQUATERNION** type.

f
Weighting factor. See Remarks.

g
Weighting factor. See Remarks.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXQuaternionBaryCentric_graphicsd3dxvb

Remarks

To compute the Barycentric coordinates, the **D3DXQuaternionBaryCentric** function implements the following series of spherical linear interpolation operations: Slerp(Slerp(Q1, Q2, f+g), Slerp(Q1, Q3, f+g), g/(f+g))

D3DXQuaternionConjugate

#Returns the conjugate of a quaternion.

D3DXQuaternionConjugate(_
QOut **As D3DQUATERNION**, _
Q **As D3DQUATERNION)**

Parameters

QOut
D3DQUATERNION type that is the result of the operation, the conjugate of the quaternion.

Q
The source **D3DQUATERNION** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Given a quaternion (x, y, z, w), the **D3DXQuaternionConjugate** function will return the quaternion (-x, -y, -z, w).

See Also

D3DXQuaternionInverse

D3DXQuaternionDot

#Returns the conjugate of a quaternion.

IDH_D3DXQuaternionConjugate_graphicsd3dxvb

D3DXQuaternionDot(_
Q1 **As D3DQUATERNION**, _
Q2 **As D3DQUATERNION)**

Parameters

Q1
A source **D3DQUATERNION** type.

Q2
A source **D3DQUATERNION** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXQuaternionExp

#Calculates the exponential.

D3DXQuaternionExp(_
QOut **As D3DQUATERNION**, _
Q **As D3DQUATERNION)**

Parameters

QOut
D3DQUATERNION type that is the result of the operation, the exponential.

Q
The source **D3DQUATERNION** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXQuaternionDot_graphicsd3dxvb
IDH_D3DXQuaternionExp_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXQuaternionExp** function works only with pure quaternions, where *w* == 0.

This function is implemented in the following manner.

```
q = (0, theta * v)
Exp(q) = (Cos(theta), Sin(theta) * v)
```

The **D3DXQuaternionExp** and **D3DXQuaternionLn** functions are useful when using the **D3DXQuaternionSquad** function. Given a set of quaternion keys (q0, q1, q2, ..., q*n*), you can compute the inner quadrangle points (a1, a2, a3, ..., a*n*-1) as shown in the following example, to insure that the tangents are continuous across adjacent segments.

```
      a1   a2   a3
 q0   q1   q2   q3   q4

a[i] = q[i] * Exp( -(ln(inv(q[i])*q[i+1]) + ln(inv(q[i])*q[i-1])) / 4 )
```

Once (a1, a2, a3, ...) are computed, you can use the results to interpolate along the curve.

```
qt = Squad(t, q[i], a[i], a[i+1], q[i+1])
```

See Also

D3DXQuaternionLn, **D3DXQuaternionSquad**

D3DXQuaternionIdentity

#Returns the identity quaternion.

D3DXQuaternionIdentity(_
QOut **As D3DQUATERNION)**

Parameters

QOut
D3DQUATERNION type that is the result of the operation, the identity quaternion.

IDH_D3DXQuaternionIdentity_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

Given a quaternion (x, y, z, w), the **D3DXQuaternionIdentity** function will return the quaternion (0, 0, 0, 1).

See Also

D3DXQuaternionIsIdentity

D3DXQuaternionInverse

#Conjugates and renormalizes a quaternion.

D3DXQuaternionInverse(_
QOut **As D3DQUATERNION**, _
Q **As D3DQUATERNION)**

Parameters

QOut
D3DQUATERNION type that is the result of the operation, the inverse quaternion of the quaternion.

Q
The source **D3DQUATERNION** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXQuaternionInverse_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXQuaternionConjugate, **D3DXQuaternionNormalize**

D3DXQuaternionIsIdentity

#Determines if a quaternion is an identity quaternion.

```
D3DXQuaternionIsIdentity( _
    Q As D3DQUATERNION) As Long
```

Parameters

Q
D3DQUATERNION type that will be tested for identity.

Return Values

If the quaternion is an identity quaternion, this function returns a non-zero value (TRUE); otherwise, this function returns zero (FALSE).

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXQuaternionIdentity

D3DXQuaternionLength

#Returns the length of a quaternion.

```
D3DXQuaternionLength( _
    Q As D3DQUATERNION) As Single
```

IDH_D3DXQuaternionIsIdentity_graphicsd3dxvb
IDH_D3DXQuaternionLength_graphicsd3dxvb

Parameters

Q
The source **D3DQUATERNION** type.

Return Values

The quaternion's length.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXQuaternionLengthSq

D3DXQuaternionLengthSq

#Returns the square of the length of a quaternion.

```
D3DXQuaternionLengthSq( _
  Q As D3DQUATERNION) As Single
```

Parameters

Q
The source **D3DQUATERNION** type.

Return Values

The quaternion's squared length.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

IDH_D3DXQuaternionLengthSq_graphicsd3dxvb

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXQuaternionLength

D3DXQuaternionLn

#Calculates the natural logarithm.

```
D3DXQuaternionLn( _
   QOut As D3DQUATERNION, _
   Q As D3DQUATERNION)
```

Parameters

QOut

D3DQUATERNION type that is the result of the operation, the natural logarithm.

Q

The source **D3DQUATERNION** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXQuaternionLn** function works only for unit quaternions.

This function is implemented in the following manner.

```
q = (Cos(theta), Sin(theta) * v)
|v| = 1
ln(q) = (0, theta * v)
```

IDH_D3DXQuaternionLn_graphicsd3dxvb

The **D3DXQuaternionExp** and **D3DXQuaternionLn** functions are useful when using the **D3DXQuaternionSquad** function. Given a set of quaternion keys (q0, q1, q2, ..., q*n*), you can compute the inner quadrangle points (a1, a2, a3, ..., a*n*-1) to insure that the tangents are continuous across adjacent segments.

```
      a1   a2   a3
  q0   q1   q2   q3   q4

a[i] = q[i] * Exp( -(ln(inv(q[i])*q[i+1]) + ln(inv(q[i])*q[i-1])) / 4 )
```

Once (a1, a2, a3, ...) are computed, you can use the results to interpolate along the curve.

```
qt = Squad(t, q[i], a[i], a[i+1], q[i+1])
```

See Also

D3DXQuaternionExp, **D3DXQuaternionSquad**

D3DXQuaternionMultiply

#Multiplies two quaternions.

```
D3DXQuaternionMultiply( _
    QOut As D3DQUATERNION, _
    Q1 As D3DQUATERNION, _
    Q2 As D3DQUATERNION)
```

Parameters

QOut

D3DQUATERNION type that is the result of the operation, the product of two quaternions.

Q1

A source **D3DQUATERNION** type.

Q2

A source **D3DQUATERNION** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXQuaternionMultiply_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The result represents the rotation Q2 followed by the rotation Q1 (Out = Q2 * Q1). The output is actually Q2*Q1 (not Q1*Q2). This is done so that **D3DXQuaternionMultiply** maintain the same semantics as **D3DXMatrixMultiply**, because unit quaternions can be considered as another way to represent rotation matrices.

Transformations are concatenated in the same order for both the **D3DXQuaternionMultiply** and **D3DXMatrixMultiply** functions. For example, assuming mX and mY represent the same rotations as qX and qY, both m and q will represent the same rotations.

```
D3DXMatrixMultiply m, mX, mY

D3DXQuaternionMultiply q, qX, qY
```

The multiplication of quaternions is not commutative.

See Also

D3DXMatrixMultiply

D3DXQuaternionNormalize

[#]Returns the normal of a quaternion.

D3DXQuaternionNormalize(_
QOut **As D3DQUATERNION**, _
Q **As D3DQUATERNION)**

Parameters

QOut
D3DQUATERNION type that is the result of the operation, the normal of the quaternion.

Q
The source **D3DQUATERNION** type.

[#] IDH_D3DXQuaternionNormalize_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXQuaternionInverse

D3DXQuaternionRotationAxis

#Rotates a quaternion about an arbitrary axis.

D3DXQuaternionRotationAxis(_
QOut **As D3DQUATERNION**, _
VAxis **As D3DVECTOR**, _
angle **As Single)**

Parameters

QOut
D3DQUATERNION type that is the result of the operation, a quaternion rotated around the specified axis.

VAxis
D3DVECTOR type that identifies the axis angle.

angle
Angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXQuaternionRotationAxis_graphicsd3dxvb

See Also

D3DXQuaternionRotationMatrix, **D3DXQuaternionRotationYawPitchRoll**

D3DXQuaternionRotationMatrix

#Builds a quaternion from a rotation matrix.

```
D3DXQuaternionRotationMatrix( _
    QOut As D3DQUATERNION, _
    M As D3DMATRIX)
```

Parameters

QOut

D3DQUATERNION type that is the result of the operation, a quaternion built from a rotation matrix.

M

The source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXQuaternionRotationAxis, **D3DXQuaternionRotationYawPitchRoll**

D3DXQuaternionRotationYawPitchRoll

#Builds a quaternion with the given yaw, pitch, and roll.

```
D3DXQuaternionRotationYawPitchRoll( _
    QOut As D3DQUATERNION, _
    yaw As Single, _
    pitch As Single, _
    roll As Single)
```

IDH_D3DXQuaternionRotationMatrix_graphicsd3dxvb
IDH_D3DXQuaternionRotationYawPitchRoll_graphicsd3dxvb

Parameters

QOut
D3DQUATERNION type that is the result of the operation, a quaternion with the specified yaw, pitch, and roll.

yaw
Yaw around the y-axis, in radians.

pitch
Pitch around the x-axis, in radians.

roll
Roll around the z-axis, in radians.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXQuaternionRotationAxis, **D3DXQuaternionRotationMatrix**

D3DXQuaternionSlerp

#Interpolates between two quaternions, using spherical linear interpolation.

```
D3DXQuaternionSlerp( _
    QOut As D3DQUATERNION, _
    Q1 As D3DQUATERNION, _
    Q2 As D3DQUATERNION, _
    t As Single)
```

Parameters

QOut
D3DQUATERNION type that is the result of the operation, the result of the interpolation.

Q1
A source **D3DQUATERNION** type.

Q2

IDH_D3DXQuaternionSlerp_graphicsd3dxvb

A source **D3DQUATERNION** type.

t

Parameter that indicates how far to interpolate between the quaternions.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXQuaternionSquad

#Interpolates between quaternions, using spherical quadrangle interpolation.

D3DXQuaternionSquad(_
QOut **As D3DQUATERNION**, _
Q1 **As D3DQUATERNION**, _
Q2 **As D3DQUATERNION**, _
Q3 **As D3DQUATERNION**, _
Q4 **As D3DQUATERNION**, _
t **As Single)**

Parameters

QOut

D3DQUATERNION type that is the result of the operation, the result of the spherical quadrangle interpolation.

Q1

A source **D3DQUATERNION** type.

Q2

A source **D3DQUATERNION** type.

Q3

A source **D3DQUATERNION** type.

Q4

A source **D3DQUATERNION** type.

t

Parameter that indicates how far to interpolate between the quaternions.

IDH_D3DXQuaternionSquad_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The function uses the following sequence of spherical linear interpolation operations: Slerp(Slerp(Q1, Q4, t), Slerp(Q2, Q3, t), 2t(1-t))

See Also

D3DXQuaternionExp, **D3DXQuaternionLn**

D3DXQuaternionToAxisAngle

#Computes a quaternion's axis and angle of rotation.

```
D3DXQuaternionToAxisAngle( _
    Q As D3DQUATERNION, _
    VAxis As D3DVECTOR, _
    angle As Single) As Long
```

Parameters

Q
The source **D3DQUATERNION** type.

VAxis
When this function returns, a **D3DVECTOR** type that identifies the quaternion's axis.

angle
When this function returns, a **Single** value that identifies the quaternion's angle of rotation, in radians.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

IDH_D3DXQuaternionToAxisAngle_graphicsd3dxvb

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function expects unit quaternions.

D3DXVec2Add

#Adds two 2-D vectors.

D3DXVec2Add(_
VOut **As D3DVECTOR2**, _
V1 **As D3DVECTOR2**, _
V2 **As D3DVECTOR2)**

Parameters

VOut
D3DVECTOR2 type that is the result of the operation, the sum of the two vectors.

V1
A source **D3DVECTOR2** type.

V2
A source **D3DVECTOR2** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec2Subtract, **D3DXVec2Scale**

IDH_D3DXVec2Add_graphicsd3dxvb

D3DXVec2BaryCentric

#Returns a point in Barycentric coordinates, using the specified 2-D vectors.

```
D3DXVec2BaryCentric( _
    VOut As D3DVECTOR2, _
    V1 As D3DVECTOR2, _
    V2 As D3DVECTOR2, _
    V3 As D3DVECTOR2, _
    f As Single, _
    g As Single)
```

Parameters

VOut
D3DVECTOR2 type that is the result of the operation, a vector in Barycentric coordinates

V1
A source **D3DVECTOR2** type.

V2
A source **D3DVECTOR2** type.

V3
A source **D3DVECTOR2** type.

f
Weighting factor. See Remarks.

g
Weighting factor. See Remarks.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXVec2BaryCentric** function provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function

IDH_D3DXVec2BaryCentric_graphicsd3dxvb

returns the resulting point by using the following equation: V1 + f(V2-V1) + g(V3-V1).

Any point in the plane V1V2V3 can be represented by the Barycentric coordinate (*f*,*g*).The parameter *f* controls how much V2 gets weighted into the result, and the parameter *g* controls how much V3 gets weighted into the result. Lastly, 1-*f*-*g* controls how much V1 gets weighted into the result.

Note the following relations.

- If (f>=0 And g>=0 And 1-f-g>=0), the point is inside the triangle V1V2V3.
- If (f==0 And g>=0 And 1-f-g>=0), the point is on the line V1V3.
- If (f>=0 And g==0 And 1-f-g>=0), the point is on the line V1V2.
- If (f>=0 And g>=0 And 1-f-g==0), the point is on the line V2V3.

Barycentric coordinates are a form of general coordinates. In this context, using Barycentric coordinates simply represents a change in coordinate systems; what holds true for Cartesian coordinates holds true for Barycentric coordinates.

D3DXVec2CatmullRom

#Performs a Catmull-Rom interpolation, using the specified 2-D vectors.

```
D3DXVec2CatmullRom( _
   VOut As D3DVECTOR2, _
   V0 As D3DVECTOR2, _
   V1 As D3DVECTOR2, _
   V2 As D3DVECTOR2, _
   V3 As D3DVECTOR2, _
   s As Single)
```

Parameters

VOut
D3DVECTOR2 type that is the result of the operation.

V0
D3DVECTOR2 type, a position vector.

V1
D3DVECTOR2 type, a position vector.

V2
D3DVECTOR2 type, a position vector.

V3
D3DVECTOR2 type, a position vector.

s
Weighting factor. See Remarks.

IDH_D3DXVec2CatmullRom_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXVec2CatmullRom** function interpolates between the position V1 when *s* is equal to 0, and between the position V2 when *s* is equal to 1, using Catmull-Rom interpolation.

D3DXVec3CatmullRom

#Performs a Catmull-Rom interpolation, using the specified 3-D vectors.

```
D3DXVec3CatmullRom( _
    VOut As D3DVECTOR, _
    V0 As D3DVECTOR, _
    V1 As D3DVECTOR, _
    V2 As D3DVECTOR, _
    V3 As D3DVECTOR, _
    s As Single)
```

Parameters

VOut
D3DVECTOR type that is the result of the operation.

V0
D3DVECTOR type, a position vector.

V1
D3DVECTOR type, a position vector.

V2
D3DVECTOR type, a position vector.

V3
D3DVECTOR type, a position vector.

s
Weighting factor. See Remarks.

IDH_D3DXVec3CatmullRom_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXVec3CatmullRom** function interpolates between the position V1 when *s* is equal to 0, and between the position V2 when *s* is equal to 1, using Catmull-Rom interpolation.

D3DXVec2CCW

#Returns the z-component by taking the cross product of two 2-D vectors.

```
D3DXVec2CCW( _
   V1 As D3DVECTOR2, _
   V2 As D3DVECTOR2) As Single
```

Parameters

V1

A source **D3DVECTOR2** type.

V2

A source **D3DVECTOR2** type.

Return Values

The z-component.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXVec2CCW_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function determines the z-component by determining the cross-product based on the following formula: ((x1,y1,0) cross (x2,y2,0)); or as shown in the following example.

```
V1.x * V2.y - V1.y * V2.x
```

If the value of the z-component is positive, the vector V2 is counter-clockwise from the vector V1. This information is useful for back-face culling.

See Also

D3DXVec2Dot

D3DXVec2Dot

#Determines the dot-product of two 2-D vectors.

```
D3DXVec2Dot( _
    V1 As D3DVECTOR2, _
    V2 As D3DVECTOR2) As Single
```

Parameters

V1
A source **D3DVECTOR2** type.

V2
A source **D3DVECTOR2** type.

Return Values

The dot-product.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXVec2Dot_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec2CCW

D3DXVec2Hermite

#Performs a Hermite spline interpolation, using the specified 2-D vectors.

```
D3DXVec2Hermite( _
    VOut As D3DVECTOR2, _
    V1 As D3DVECTOR2, _
    T1 As D3DVECTOR2, _
    V2 As D3DVECTOR2, _
    T2 As D3DVECTOR2, _
    s As Single)
```

Parameters

VOut
D3DVECTOR2 type that is the result of the operation, the Hermite spline interpolation.

V1
A source **D3DVECTOR2** type, a position vector.

T1
A source **D3DVECTOR2** type, a tangent vector.

V2
A source **D3DVECTOR2** type, a position vector.

T2
A source **D3DVECTOR2** type, a tangent vector.

s
Weighting factor. See Remarks.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXVec2Hermite_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXVec2Hermite** function interpolates from (positionA, tangentA) to (positionB, tangentB) using Hermite spline interpolation. This function interpolates between the position V1 and the tangent T1, when *s* is equal to 0, and between the position V2 and the tangent T2, when *s* is equal to 1.

Hermite splines are useful for controlling animation because the curve runs through all the control points. Also, because the position and tangent are explicitly specified at the ends of each segment, it is easy to create a C2 continuous curve as long as you make sure that your starting position and tangent match the ending values of the last segment.

D3DXVec2Length

#Returns the length of a 2-D vector.

```
D3DXVec2Length( _
    v As D3DVECTOR2) As Single
```

Parameters

v

The source **D3DVECTOR2** type.

Return Values

The vector's length.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec2LengthSq

IDH_D3DXVec2Length_graphicsd3dxvb

D3DXVec2LengthSq

#Returns the square of the length of a 2-D vector.

```
D3DXVec2LengthSq( _
   v As D3DVECTOR2) As Single
```

Parameters

v

The source **D3DVECTOR2** type.

Return Values

The vector's squared length.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec2Length

D3DXVec2Lerp

#Performs a linear interpolation between two 2-D vectors.

```
D3DXVec2Lerp( _
   VOut As D3DVECTOR2, _
   V1 As D3DVECTOR2, _
   V2 As D3DVECTOR2, _
   s As Single)
```

Parameters

VOut

IDH_D3DXVec2LengthSq_graphicsd3dxvb
IDH_D3DXVec2Lerp_graphicsd3dxvb

D3DVECTOR2 type that is the result of the operation, the linear interpolation.

V1

A source **D3DVECTOR2** type.

V2

A source **D3DVECTOR2** type.

s

Parameter that linearly interpolates between the vectors.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXVec2Maximize

#Returns a 2-D vector that is made up of the largest components of two 2-D vectors.

D3DXVec2Maximize(_
VOut **As D3DVECTOR2**, _
V1 **As D3DVECTOR2**, _
V2 **As D3DVECTOR2)**

Parameters

VOut

D3DVECTOR2 type that is the result of the operation, a vector made up of the largest components of the two vectors.

V1

A source **D3DVECTOR2** type.

V2

A source **D3DVECTOR2** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXVec2Maximize_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec2Minimize

D3DXVec2Minimize

#Returns a 2-D vector that is made up of the smallest components of two 2-D vectors.

D3DXVec2Minimize(_
VOut **As D3DVECTOR2**, _
V1 **As D3DVECTOR2**, _
V2 **As D3DVECTOR2)**

Parameters

VOut
D3DVECTOR2 type that is the result of the operation, a vector made up of the smallest components of the two vectors.

V1
A source **D3DVECTOR2** type.

V2
A source **D3DVECTOR2** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec2Maximize

IDH_D3DXVec2Minimize_graphicsd3dxvb

D3DXVec2Normalize

#Returns the normalized version of a 2-D vector.

```
D3DXVec2Normalize( _
    VOut As D3DVECTOR2, _
    v As D3DVECTOR2)
```

Parameters

VOut

D3DVECTOR2 type that is the result of the operation, the normalized version of the vector.

v

The source **D3DVECTOR2** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXVec2Scale

#Scales a 2-D vector.

```
D3DXVec2Scale( _
    VOut As D3DVECTOR2, _
    V1 As D3DVECTOR2, _
    s As Single)
```

Parameters

VOut

D3DVECTOR2 type that is the result of the operation, the scaled vector.

V1

The source **D3DVECTOR2** type.

s

Scaling value.

IDH_D3DXVec2Normalize_graphicsd3dxvb
IDH_D3DXVec2Scale_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec2Add, **D3DXVec2Subtract**

D3DXVec2Subtract

#Subtracts two 2-D vectors.

D3DXVec2Subtract(_
VOut **As D3DVECTOR2**, _
V1 **As D3DVECTOR2**, _
V2 **As D3DVECTOR2)**

Parameters

VOut
D3DVECTOR2 type that is the result of the operation, the difference of two vectors.

V1
A source **D3DVECTOR2** type.

V2
A source **D3DVECTOR2** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXVec2Subtract_graphicsd3dxvb

See Also

D3DXVec2Add, **D3DXVec2Scale**

D3DXVec2Transform

#Transforms a 2-D vector by a given matrix.

```
D3DXVec2Transform( _
    VOut As D3DVECTOR4, _
    V1 As D3DVECTOR2, _
    M As D3DMATRIX)
```

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, the transformed vector.

V1
The source **D3DVECTOR2** type.

M
The source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function transform the vector, *V1* (x, y, 0, 1), by the matrix, *M*.

See Also

D3DXVec2TransformCoord, **D3DXVec2TransformNormal**

D3DXVec2TransformCoord

#Transforms a 2-D vector by a given matrix, projecting the result back into w = 1.

IDH_D3DXVec2Transform_graphicsd3dxvb
IDH_D3DXVec2TransformCoord_graphicsd3dxvb

D3DXVec2TransformCoord(_
VOut **As D3DVECTOR2**, _
V1 **As D3DVECTOR2**, _
M **As D3DMATRIX)**

Parameters

VOut
D3DVECTOR2 type that is the result of the operation, the transformed vector.

V1
The source **D3DVECTOR2** type.

M
The source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function transform the vector, *V1* (x, y, 0, 1), by the matrix, *M*, projecting the result back into w=1.

See Also

D3DXVec2Transform, **D3DXVec2TransformNormal**

D3DXVec2TransformNormal

[#]Transforms the 2-D vector normal by the given matrix.

D3DXVec2TransformNormal(_
VOut **As D3DVECTOR2**, _
V1 **As D3DVECTOR2**, _
M **As D3DMATRIX)**

[#] IDH_D3DXVec2TransformNormal_graphicsd3dxvb

Parameters

VOut
D3DVECTOR2 type that is the result of the operation, the transformed vector.

V1
The source **D3DVECTOR2** type.

M
The source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function transforms the vector normal (x, y, 0, 0) of the vector, *V1*, by the matrix, *M*.

See Also

D3DXVec2Transform, **D3DXVec2TransformCoord**

D3DXVec3Add

#Adds two 3-D vectors.

```
D3DXVec3Add( _
   VOut As D3DVECTOR, _
   V1 As D3DVECTOR, _
   V2 As D3DVECTOR)
```

Parameters

VOut
D3DVECTOR type that is the result of the operation, the sum of the two 3-D vectors.

V1
A source **D3DVECTOR** type.

IDH_D3DXVec3Add_graphicsd3dxvb

V2
A source **D3DVECTOR** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec3Subtract, **D3DXVec3Scale**

D3DXVec3BaryCentric

#Returns a point in Barycentric coordinates, using the specified 3-D vectors.

```
D3DXVec3BaryCentric( _
    VOut As D3DVECTOR, _
    V1 As D3DVECTOR, _
    V2 As D3DVECTOR, _
    V3 As D3DVECTOR, _
    f As Single, _
    g As Single)
```

Parameters

VOut
D3DVECTOR type that is the result of the operation, a vector in Barycentric coordinates.

V1
A source **D3DVECTOR** type.

V2
A source **D3DVECTOR** type.

V3
A source **D3DVECTOR** type.

f
Weighting factor. See Remarks.

g

IDH_D3DXVec3BaryCentric_graphicsd3dxvb

Weighting factor. See Remarks.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXVec3BaryCentric** function provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function returns the resulting point by using the following equation: V1 + f(V2-V1) + g(V3-V1).

Any point in the plane V1V2V3 can be represented by the Barycentric coordinate (*f*,*g*).The parameter *f* controls how much V2 gets weighted into the result, and the parameter *g* controls how much V3 gets weighted into the result. Lastly, 1-*f*-*g* controls how much V1 gets weighted into the result.

Note the following relations.

- If (f>=0 And g>=0 And 1-f-g>=0), the point is inside the triangle V1V2V3.
- If (f==0 And g>=0 And 1-f-g>=0), the point is on the line V1V3.
- If (f>=0 And g==0 And 1-f-g>=0), the point is on the line V1V2.
- If (f>=0 And g>=0 And 1-f-g==0), the point is on the line V2V3.

Barycentric coordinates are a form of general coordinates. In this context, using Barycentric coordinates simply represents a change in coordinate systems; what holds true for Cartesian coordinates holds true for Barycentric coordinates.

D3DXVec3Cross

#Determines the cross-product of two 3-D vectors.

D3DXVec3Cross(_
VOut **As D3DVECTOR**, _
V1 **As D3DVECTOR**, _
V2 **As D3DVECTOR)**

IDH_D3DXVec3Cross_graphicsd3dxvb

Parameters

VOut
D3DVECTOR type that is the result of the operation, the cross product of two 3-D vectors.

V1
A source **D3DVECTOR** type.

V2
A source **D3DVECTOR** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function determines the cross-product with the following code.

```
Dim VOut As D3DVECTOR

VOut.x = V1.y * V2.z - V1.z * V2.y
VOut.y = V1.z * V2.x - V1.x * V2.z
VOut.z = V1.x * V2.y - V1.y * V2.x
```

See Also

D3DXVec3Dot

D3DXVec3Dot

#Determines the dot-product of two 3-D vectors.

```
D3DXVec3Dot( _
   V1 As D3DVECTOR, _
   V2 As D3DVECTOR) As Single
```

IDH_D3DXVec3Dot_graphicsd3dxvb

Parameters

V1
A source **D3DVECTOR** type.

V2
A source **D3DVECTOR** type.

Return Values

The dot-product.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec3Cross

D3DXVec3Hermite

#Performs a Hermite spline interpolation, using the specified 3-D vectors.

```
D3DXVec3Hermite( _
    VOut As D3DVECTOR, _
    V1 As D3DVECTOR, _
    T1 As D3DVECTOR, _
    V2 As D3DVECTOR, _
    T2 As D3DVECTOR, _
    s As Single)
```

Parameters

VOut
D3DVECTOR type that is the result of the operation, the Hermite spline interpolation.

V1
A source **D3DVECTOR** type, a position vector.

IDH_D3DXVec3Hermite_graphicsd3dxvb

T1
A source **D3DVECTOR** type, a tangent vector.

V2
A source **D3DVECTOR** type, a position vector.

T2
A source **D3DVECTOR** type, a tangent vector.

s
Weighting factor. See Remarks.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXVec3Hermite** function interpolates from (positionA, tangentA) to (positionB, tangentB) using Hermite spline interpolation. This function interpolates between the position V1 and the tangent T1, when *s* is equal to zero, and between the position V2 and the tangent T2, when *s* is equal to one.

Hermite splines are useful for controlling animation because the curve runs through all the control points. Also, because the position and tangent are explicitly specified at the ends of each segment, it is easy to create a C2 continuous curve as long as you make sure that your starting position and tangent match the ending values of the last segment.

D3DXVec3Length

#Returns the length of a 3-D vector.

```
D3DXVec3Length( _
   v As D3DVECTOR) As Single
```

Parameters

v
The source **D3DVECTOR** type.

IDH_D3DXVec3Length_graphicsd3dxvb

Return Values

The vector's length.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec3LengthSq

D3DXVec3LengthSq

#Returns the square of the length of a 3-D vector.

```
D3DXVec3LengthSq( _
   v As D3DVECTOR) As Single
```

Parameters

v

The source **D3DVECTOR** type.

Return Values

The vector's squared length.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXVec3LengthSq_graphicsd3dxvb

See Also

D3DXVec3Length

D3DXVec3Lerp

#Performs a linear interpolation between two 3-D vectors.

```
D3DXVec3Lerp( _
    VOut As D3DVECTOR, _
    V1 As D3DVECTOR, _
    V2 As D3DVECTOR, _
    s As Single)
```

Parameters

VOut
D3DVECTOR type that is the result of the operation, the linear interpolation.

V1
A source **D3DVECTOR** type.

V2
A source **D3DVECTOR** type.

s
Parameter that linearly interpolates between the vectors.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function performs the linear interpolation based on the following formula: V1 + s(V2-V1).

D3DXVec3Maximize

#Returns a 3-D vector that is made up of the largest components of two 3-D vectors.

IDH_D3DXVec3Lerp_graphicsd3dxvb
IDH_D3DXVec3Maximize_graphicsd3dxvb

D3DXVec3Maximize(_
VOut **As D3DVECTOR**, _
V1 **As D3DVECTOR**, _
V2 **As D3DVECTOR)**

Parameters

VOut
D3DVECTOR type that is the result of the operation, a vector made up of the largest components of the two vectors.

V1
A source **D3DVECTOR** type.

V2
A source **D3DVECTOR** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec3Minimize

D3DXVec3Minimize

#Returns a 3-D vector that is made up of the smallest components of two 3-D vectors.

D3DXVec3Minimize(_
VOut **As D3DVECTOR**, _
V1 **As D3DVECTOR**, _
V2 **As D3DVECTOR)**

Parameters

VOut
D3DVECTOR type that is the result of the operation, a vector made up of the smallest components of the two vectors.

IDH_D3DXVec3Minimize_graphicsd3dxvb

V1
A source **D3DVECTOR** type.

V2
A source **D3DVECTOR** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec3Maximize

D3DXVec3Normalize

#Returns the normalized version of a 3-D vector.

```
D3DXVec3Normalize( _
    VOut As D3DVECTOR, _
    v As D3DVECTOR)
```

Parameters

VOut
D3DVECTOR type that is the result of the operation, the normalized version of the specified vector.

v
The source **D3DVECTOR** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXVec3Normalize_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXVec3Project

[#]Projects a vector from object space into screen space.

```
D3DXVec3Project( _
    VOut As D3DVECTOR, _
    V As D3DVECTOR, _
    Viewport As D3DVIEWPORT8, _
    Projection As D3DMATRIX, _
    View As D3DMATRIX, _
    World As D3DMATRIX)
```

Parameters

VOut
D3DVECTOR type that is the result of the operation, the vector projected from object space to screen space.

V
The source **D3DVECTOR** type.

Viewport
D3DVIEWPORT8 type, representing the viewport.

Projection
D3DMATRIX type, representing the projection matrix.

View
D3DMATRIX type, representing the view matrix.

World
D3DMATRIX type, representing the world matrix.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

[#] IDH_D3DXVec3Project_graphicsd3dxvb

See Also

D3DXVec3Unproject

D3DXVec3Scale

#Scales a 3-D vector.

```
D3DXVec3Scale( _
    VOut As D3DVECTOR2, _
    V1 As D3DVECTOR2, _
    s As Single)
```

Parameters

VOut
D3DVECTOR type that is the result of the operation, the scaled vector.

V1
A source **D3DVECTOR** type.

s
Scaling value.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec3Add, **D3DXVec3Subtract**

D3DXVec3Subtract

#Subtracts two 3-D vectors.

```
D3DXVec3Subtract( _
    VOut As D3DVECTOR, _
    V1 As D3DVECTOR, _
```

IDH_D3DXVec3Scale_graphicsd3dxvb
IDH_D3DXVec3Subtract_graphicsd3dxvb

V2 **As D3DVECTOR)**

Parameters

VOut
D3DVECTOR type that is the result of the operation, the difference of two vectors.

V1
A source **D3DVECTOR** type.

V2
A source **D3DVECTOR** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec3Add, **D3DXVec3Scale**

D3DXVec3Transform

#Transforms a 3-D vector by a given matrix.

D3DXVec3Transform(_
VOut **As D3DVECTOR4**, _
V1 **As D3DVECTOR**, _
M **As D3DMATRIX)**

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, the transformed vector.

V1
The source **D3DVECTOR** type.

M
The source **D3DMATRIX** type.

IDH_D3DXVec3Transform_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function transform the vector, *V1* (x, y, z, 1), by the matrix, *M*.

See Also

D3DXVec3TransformCoord, **D3DXVec3TransformNormal**

D3DXVec3TransformCoord

#Transforms a 3-D vector by a given matrix, projecting the result back into w = 1.

D3DXVec3TransformCoord(_
VOut **As D3DVECTOR**, _
V1 **As D3DVECTOR**, _
M **As D3DMATRIX)**

Parameters

VOut
D3DVECTOR type that is the result of the operation, the transformed vector.

V1
The source **D3DVECTOR** type.

M
The source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXVec3TransformCoord_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function transform the vector, *V1* (x, y, z, 1), by the matrix, *M*, projecting the result back into w=1.

See Also

D3DXVec3Transform, **D3DXVec3TransformNormal**

D3DXVec3TransformNormal

#Transforms the 3-D vector normal by the given matrix.

D3DXVec3TransformNormal(_
VOut **As D3DVECTOR, _**
V1 **As D3DVECTOR, _**
M **As D3DMATRIX)**

Parameters

VOut
D3DVECTOR type that is the result of the operation, the transformed vector.

V1
The source **D3DVECTOR** type.

M
The source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function transforms the vector normal (x, y, z, 0) of the vector, *V1*, by the matrix, *M*.

IDH_D3DXVec3TransformNormal_graphicsd3dxvb

If you transform a normal by a non-affine matrix, the matrix you pass to this function should be the transpose of the inverse of the matrix you would use to transform a coordinate.

D3DXVec3Unproject

#Projects a vector from screen space into object space.

```
D3DXVec3Unproject( _
    VOut As D3DVECTOR, _
    V As D3DVECTOR, _
    Viewport As D3DVIEWPORT8, _
    Projection As D3DMATRIX, _
    View As D3DMATRIX, _
    World As D3DMATRIX)
```

Parameters

VOut
D3DVECTOR type that is the result of the operation, the vector projected from screen space to object space.

V
The source **D3DVECTOR** type.

Viewport
D3DVIEWPORT8 type, representing the viewport.

Projection
D3DMATRIX type, representing the projection matrix.

View
D3DMATRIX type, representing the view matrix.

World
D3DMATRIX type, representing the world matrix.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

IDH_D3DXVec3Unproject_graphicsd3dxvb

See Also

D3DXVec3Project

D3DXVec4Add

#Adds two 4-D vectors.

D3DXVec4Add(_
VOut **As D3DVECTOR4**, _
V1 **As D3DVECTOR4**, _
V2 **As D3DVECTOR4)**

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, the sum of the two vectors.

V1
A source **D3DVECTOR4** type.

V2
A source **D3DVECTOR4** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec4Subtract, **D3DXVec4Scale**

D3DXVec4BaryCentric

#Returns a point in Barycentric coordinates, using the specified 4-D vectors.

D3DXVec4BaryCentric(_
VOut **As D3DVECTOR4**, _

IDH_D3DXVec4Add_graphicsd3dxvb
IDH_D3DXVec4BaryCentric_graphicsd3dxvb

V1 **As D3DVECTOR4**, _
V2 **As D3DVECTOR4**, _
V3 **As D3DVECTOR4**, _
f **As Single**, _
g **As Single)**

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, a vector in Barycentric coordinates.

V1
A source **D3DVECTOR4** type.

V2
A source **D3DVECTOR4** type.

V3
A source **D3DVECTOR4** type.

f
Weighting factor. See Remarks.

g
Weighting factor. See Remarks.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXVec4BaryCentric** function provides a way to understand points in and around a triangle, independent of where the triangle is actually located. This function returns the resulting point by using the following equation: V1 + f(V2-V1) + g(V3-V1).

Any point in the plane V1V2V3 can be represented by the Barycentric coordinate (*f*,*g*).The parameter *f* controls how much V2 gets weighted into the result, and the parameter *g* controls how much V3 gets weighted into the result. Lastly, 1-*f*-*g* controls how much V1 gets weighted into the result.

Note the following relations.

- If (f>=0 And g>=0 And 1-f-g>=0), the point is inside the triangle V1V2V3.
- If (f==0 And g>=0 And 1-f-g>=0), the point is on the line V1V3.
- If (f>=0 And g==0 And 1-f-g>=0), the point is on the line V1V2.
- If (f>=0 And g>=0 And 1-f-g==0), the point is on the line V2V3.

Barycentric coordinates are a form of general coordinates. In this context, using Barycentric coordinates simply represents a change in coordinate systems; what holds true for Cartesian coordinates holds true for Barycentric coordinates.

D3DXVec4CatmullRom

#Performs a Catmull-Rom interpolation, using the specified 4-D vectors.

D3DXVec4CatmullRom(_
VOut **As D3DVECTOR4**, _
V0 **As D3DVECTOR4**, _
V1 **As D3DVECTOR4**, _
V2 **As D3DVECTOR4**, _
V3 **As D3DVECTOR4**, _
s **As Single)**

Parameters

VOut
D3DVECTOR4 type that is the result of the operation.

V0
D3DVECTOR4 type, a position vector.

V1
D3DVECTOR4 type, a position vector.

V2
D3DVECTOR4 type, a position vector.

V3
D3DVECTOR4 type, a position vector.

s
Weighting factor. See Remarks.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXVec4CatmullRom_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXVec4CatmullRom** function interpolates between the position V1 when *s* is equal to 0, and between the position V2 when *s* is equal to 1, using Catmull-Rom interpolation.

D3DXVec4Cross

#Determines the cross-product in 4 dimensions.

```
D3DXVec4Cross( _
    VOut As D3DVECTOR4, _
    V1 As D3DVECTOR4, _
    V2 As D3DVECTOR4, _
    V3 As D3DVECTOR4)
```

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, the cross product.

V1
A source **D3DVECTOR4** type.

V2
A source **D3DVECTOR4** type.

V3
A source **D3DVECTOR4** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec4Dot

IDH_D3DXVec4Cross_graphicsd3dxvb

D3DXVec4Dot

#Determines the dot-product of two 4-D vectors.

```
D3DXVec4Dot( _
   V1 As D3DVECTOR4, _
   V2 As D3DVECTOR4) As Single
```

Parameters

V1

A source **D3DVECTOR4** type.

V2

A source **D3DVECTOR4** type.

Return Values

The dot-product.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec4Cross

D3DXVec4Hermite

#Performs a Hermite spline interpolation, using the specified 4-D vectors.

```
D3DXVec4Hermite( _
   VOut As D3DVECTOR4, _
   V1 As D3DVECTOR, _
   T1 As D3DVECTOR, _
   V2 As D3DVECTOR, _
   T2 As D3DVECTOR, _
   s As Single)
```

IDH_D3DXVec4Dot_graphicsd3dxvb
IDH_D3DXVec4Hermite_graphicsd3dxvb

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, the Hermite spline interpolation.

V1
A source **D3DVECTOR** type, a position vector.

T1
A source **D3DVECTOR4** type, a tangent vector.

V2
A source **D3DVECTOR** type, a position vector.

T2
A source **D3DVECTOR4** type, a tangent vector.

s
Weighting factor. See Remarks.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

The **D3DXVec4Hermite** function interpolates from (positionA, tangentA) to (positionB, tangentB) using Hermite spline interpolation. This function interpolates between the position V1 and the tangent T1, when *s* is equal to zero, and between the position V2 and the tangent T2, when *s* is equal to one.

Hermite splines are useful for controlling animation because the curve runs through all the control points. Also, because the position and tangent are explicitly specified at the ends of each segment, it is easy to create a C2 continuous curve as long as you make sure that your starting position and tangent match the ending values of the last segment.

D3DXVec4Length

#Returns the length of a 4-D vector.

IDH_D3DXVec4Length_graphicsd3dxvb

```
D3DXVec4Length( _
   v As D3DVECTOR4) As Single
```

Parameters

v
A source **D3DVECTOR4** type.

Return Values

The vector's length.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec4LengthSq

D3DXVec4LengthSq

#Returns the square of the length of a 4-D vector.

```
D3DXVec4LengthSq( _
   v As D3DVECTOR4) As Single
```

Parameters

v
A source **D3DVECTOR4** type.

Return Values

The vector's squared length.

IDH_D3DXVec4LengthSq_graphicsd3dxvb

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec4Length

D3DXVec4Lerp

#Performs a linear interpolation between two 4-D vectors.

D3DXVec4Lerp(_
VOut **As D3DVECTOR4**, _
V1 **As D3DVECTOR4**, _
V2 **As D3DVECTOR4**, _
s **As Single)**

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, the linear interpolation.

V1
A source **D3DVECTOR4** type.

V2
A source **D3DVECTOR4** type.

s
Parameter that linearly interpolates between the vectors.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

IDH_D3DXVec4Lerp_graphicsd3dxvb

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Remarks

This function performs the linear interpolation based on the following formula: V1 + s(V2-V1).

D3DXVec4Maximize

#Returns a 4-D vector that is made up of the largest components of two 4-D vectors.

D3DXVec4Maximize(_
VOut **As D3DVECTOR4**, _
V1 **As D3DVECTOR4**, _
V2 **As D3DVECTOR4)**

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, a vector made up of the largest components of the two vectors.

V1
A source **D3DVECTOR4** type.

V2
A source **D3DVECTOR4** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec4Minimize

D3DXVec4Minimize

#Returns a 4-D vector that is made up of the smallest components of two 4-D vectors.

IDH_D3DXVec4Maximize_graphicsd3dxvb
IDH_D3DXVec4Minimize_graphicsd3dxvb

D3DXVec4Minimize(_
VOut **As D3DVECTOR4**, _
V1 **As D3DVECTOR4**, _
V2 **As D3DVECTOR4)**

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, a vector made up of the smallest components of the two vectors.

V1
A source **D3DVECTOR4** type.

V2
A source **D3DVECTOR4** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec4Maximize

D3DXVec4Normalize

#Returns the normalized version of a 4-D vector.

D3DXVec4Normalize(_
VOut **As D3DVECTOR4**, _
v **As D3DVECTOR4)**

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, the normalized version of the vector.

v

IDH_D3DXVec4Normalize_graphicsd3dxvb

The source **D3DVECTOR4** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

D3DXVec4Scale

#Scales a 4-D vector.

D3DXVec4Scale(_
VOut **As D3DVECTOR4**, **_**
V1 **As D3DVECTOR4**, **_**
s **As Single)**

Parameters

VOut

D3DVECTOR4 type that is the result of the operation, the scaled vector.

V1

The source **D3DVECTOR4** type.

s

Scaling value.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec4Add, **D3DXVec4Subtract**

IDH_D3DXVec4Scale_graphicsd3dxvb

D3DXVec4Subtract

#Subtracts two 4-D vectors.

```
D3DXVec4Subtract( _
    VOut As D3DVECTOR4, _
    V1 As D3DVECTOR4, _
    V2 As D3DVECTOR4)
```

Parameters

VOut

D3DVECTOR4 type that is the result of the operation, the difference of two vectors.

V1

A source **D3DVECTOR4** type.

V2

A source **D3DVECTOR4** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

See Also

D3DXVec3Add, **D3DXVec3Scale**

D3DXVec4Transform

#Transforms a 4-D vector by a given matrix.

```
D3DXVec4Transform( _
    VOut As D3DVECTOR4, _
    V1 As D3DVECTOR4, _
    M As D3DMATRIX)
```

IDH_D3DXVec4Subtract_graphicsd3dxvb
IDH_D3DXVec4Transform_graphicsd3dxvb

Parameters

VOut
D3DVECTOR4 type that is the result of the operation, the transformed 4-D vector.

V1
A source **D3DVECTOR4** type.

M
A source **D3DMATRIX** type.

Error Codes

If the function fails, an error is raised and **Err.Number** can be set to one of the following values:

D3DERR_INVALIDCALL

D3DERR_OUTOFVIDEOMEMORY

For information on trapping errors, see the Microsoft® Visual Basic® Error Handling topic.

Types

This section contains information about the types included with the Direct3DX utility library.

- **D3DXATTRIBUTERANGE**
- **D3DXATTRIBUTEWEIGHTS**
- **D3DXBONECOMBINATION**
- **D3DXDECLARATOR**
- **D3DXIMAGE_INFO**
- **D3DXMATERIAL**
- **D3DXRTS_DESC**

D3DXATTRIBUTERANGE

#Stores an attribute table entry.

```
Type D3DXATTRIBUTERANGE
    AttribId As Long
    FaceCount As Long
```

IDH_D3DXATTRIBUTERANGE_graphicsd3dxvb

```
    FaceStart As Long
    VertexCount As Long
    VertexStart As Long
End Type
```

Members

AttribId
Attribute table identifier.

FaceCount
Face count.

FaceStart
Starting face.

VertexCount
Vertex count.

VertexStart
Starting vertex.

Remarks

An attribute table is used to identify areas of the mesh that need to be drawn with different textures, render states, materials, and so on. In addition, the application can use the attribute table to hide portions of a mesh by not drawing a given attribute identifier (**AttribId**) when drawing the frame.

See Also

D3DXBaseMesh.DrawSubset

D3DXATTRIBUTEWEIGHTS

#Specifies how to weigh vertex components.

```
Type D3DXATTRIBUTEWEIGHTS
    Boundary As Single
    diffuse As Single
    Normal As Single
    position As Single
    specular As Single
    Tex(0 To 7) As Single
End Type
```

IDH_D3DXATTRIBUTEWEIGHTS_graphicsd3dxvb

Members

Boundary
Boundary component weight.

diffuse
Diffuse component weight.

Normal
Normal component weight.

position
Position component weight.

specular
Specular component weight.

Tex
Texture coordinate weights.

Remarks

This type describes how a simplification operation will consider vertex data when calculating relative costs between collapsing edges. For example, if the Normal field is 0.0, then the simplification operation will ignore the vertex normal component when calculating the error for the collapse. However, if the Normal field is 1.0, then the simplification operation will use the vertex normal component. if the Normal field is 2.0, then double the amount of errors; if the Normal field is 4.0, then quadruple the number of errors, and so on.

See Also

D3DX8.CreateSPMesh, D3DX8.GeneratePMesh, D3DX8.SimplifyMesh

D3DXBONECOMBINATION

#Describes a subset of the mesh that has the same attribute and bone combination.

```
Type D3DXBONECOMBINATION
    AttribId As Long
    FaceCount As Long
    FaceStart As Long
    VertexCount As Long
    VertexStart As Long
End Type
```

Members

AttribId

IDH_D3DXBONECOMBINATION_graphicsd3dxvb

Attribute table identifier.

FaceCount
Face count.

FaceStart
Starting face.

VertexCount
Vertex count.

VertexStart
Starting vertex.

Remarks

The subset of the mesh described by D3DXBONECOMBINATION can be rendered in a single drawing call.

D3DXDECLARATOR

#Stores a declarator array.

```
Type D3DXDECLARATOR
   value(0 To 31) As Long
End Type
```

Members

value
A declarator array, describing the vertex format of vertices

See Also

D3DX8.DeclaratorFromFVF, **D3DX8.FVFFromDeclarator**

D3DXIMAGE_INFO

#Use to return a description of what the original contents of an image file looked like.

```
Type D3DXIMAGE_INFO
   Depth As Long
   Height As Long
   MipLevels As Long
   Width As Long
   Format As Long
```

IDH_D3DXDECLARATOR_graphicsd3dxvb
IDH_D3DXIMAGE_INFO_graphicsd3dxvb

```
End Type
```

Members

Depth
Width of original image in pixels.

Height
Height of original image in pixels.

MipLevels
Depth of original image in pixels

Width
Number of mip levels in original image.

Format
A value from the **CONST_D3DFORMAT** enumerated type that most closely describes the data in the original image.

See Also

D3DX8.DeclaratorFromFVF, **D3DX8.FVFFromDeclarator**

D3DXMATERIAL

#Returns material information saved in Microsoft® DirectX® (.x) files.

```
Type D3DXMATERIAL
   Material As D3DMATERIAL8
   TextureFilename As String
End Type
```

Members

Material
D3DMATERIAL8 type that describes the material properties.

TextureFilename
String that specifies the file name of the texture.

Remarks

The **D3DX8.LoadMeshFromX** and **D3DX8.LoadMeshFromXof** methods return an array of **D3DXMATERIAL** types that specify the material color and name of the texture for each material in the mesh. The application is then required to load the texture.

IDH_D3DXMATERIAL_graphicsd3dxvb

See Also

D3DX8.LoadMeshFromX, **D3DX8.LoadMeshFromXof**

D3DXRTS_DESC

#Describes a render surface.

```
Type D3DXRTS_DESC
   Width As Long
   Height As Long
   format As Long
   DepthStencil As Long
   DepthStencilFormat As Long
End Type
```

Members

Width
Width of the render surface, in pixels

Height
Height of the render surface, in pixels.

format
Member of the **CONST_3DFORMAT** enumerated type, describing the pixel format of the render surface.

DepthStencil
If True, the render surface supports a depth-stencil surface; otherwise this member is set to False.

DepthStencilFormat
If **DepthStencil** is set to True, this parameter is a member of the **D3DFORMAT** enumerated type, describing the depth-stencil format of the render surface.

See Also

D3DXRenderToSurface.GetDesc

Enumerations

This section contains information about the enumerations included with the Direct3DX utility library.

- **CONST_D3DXASM**

IDH_D3DXRTS_DESC_graphicsd3dxvb

- **CONST_D3DXENUM**
- **CONST_D3DXERR**
- **CONST_D3DXMESH**
- **CONST_D3DXMESHMISC**
- **CONST_D3DXMESHOPT**
- **CONST_D3DXMESHSIMP**
- **CONST_DTFLAGS**

CONST_D3DXASM

#Defines shader assembly flags.

```
Enum CONST_D3DXMESHMISC
   D3DXASM_DEBUG         = 1
   D3DXASM_SKIPVALIDATION = 2
End Enum
```

Constants

D3DXASM_DEBUG
: Inserts debugging information as comments in the assembled shader.

D3DXASM_SKIPVALIDATION
: Do not validate the generated code against known capabilities and constraints. This option is only recommended when assembling a shader you know will work (that is, the shader has been assembled before without this option.)

See Also

D3DX8.AssembleShader, **D3DX8.AssembleShaderFromFile**

CONST_D3DXENUM

#Defines miscellaneous Direct3DX flags, including image filters.

```
Enum CONST_D3DXENUM
   D3DX_FILTER_NONE     =     0
   D3DX_FILTER_POINT    =     1
   D3DX_FILTER_LINEAR   =     2
   D3DX_FILTER_TRIANGLE =     3
   D3DX_FILTER_BOX      =     4
   D3DX_FILTER_MIRROR_U =  65536 (&H10000)
```

IDH_CONST_D3DXASM_graphicsd3dxvb
IDH_CONST_D3DXENUM_graphicsd3dxvb

```
    D3DX_FILTER_MIRROR_V = 131072 (&H20000)
    D3DX_FILTER_MIRROR   = 196608 (&H30000)
    D3DX_FILTER_DITHER   = 524288 (&H80000)
    D3DX_DEFAULT         =     -1 (&HFFFFFFFF)
End Enum
```

Constants

D3DX_FILTER_NONE
: No scaling or filtering will take place. Pixels outside the bounds of the source image are assumed to be transparent black.

D3DX_FILTER_POINT
: Each destination pixel is computed by sampling the nearest pixel from the source image.

D3DX_FILTER_LINEAR
: Each destination pixel is computed by linearly interpolating between the nearest pixels in the source image. This filter works best when the scale on each axis is less than 2.

D3DX_FILTER_TRIANGLE
: Every pixel in the source image contributes equally to the destination image. This is the slowest of the filters.

D3DX_FILTER_BOX
: Each pixel is computed by averaging a 2×2(×2) box of pixels from the source image. This filter works only when the dimensions of the destination are half those of the source, as is the case with mipmaps.

D3DX_FILTER_MIRROR_U
: Indicates that the pixels off the edge of the texture on the u-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR_V
: Indicates that the pixels off the edge of the texture on the v-axis should be mirrored, not wrapped.

D3DX_FILTER_MIRROR
: Specifying this flag is the same as specifying both the D3DX_FILTER_MIRROR_U and D3DX_FILTER_MIRROR_V flags.

D3DX_FILTER_DITHER
Dither the resulting image using a 4x4 ordered dither algorithm.D3DX_DEFAULT
: Predefined value that can be used for a parameter that is an enumerated value or a handle. The documentation indicates wherever D3DX_DEFAULT may be used, and how it will be interpreted in each situation.

Remarks

Each valid filter must contain exactly one of the following flags.

- D3DX_FILTER_BOX

- D3DX_FILTER_LINEAR
- D3DX_FILTER_NONE
- D3DX_FILTER_POINT
- D3DX_FILTER_TRIANGLE

In addition, you can use the **OR** operator to specify zero or more of the following optional flags with a valid filter.

- D3DX_FILTER_MIRROR
- D3DX_FILTER_MIRROR_U
- D3DX_FILTER_MIRROR_V

See Also

D3DX8.FilterCubeTexture, **D3DX8.FilterTexture**, **D3DX8.LoadSurfaceFromFile**, **D3DX8.LoadSurfaceFromFileInMemory**, **D3DX8.LoadSurfaceFromMemory**, **D3DX8.LoadSurfaceFromSurface**

CONST_D3DXERR

#Defines error codes raised by the system. For descriptions of these error codes, see Error Codes.

CONST_D3DXMESH

#Defines creation options for a mesh.

```
Enum CONST_D3DXMESH
   D3DXMESH_32BIT       =       1
   D3DXMESH_DONOTCLIP   =       2
   D3DXMESH_POINTS      =       4
   D3DXMESH_RTPATCHES   =       8
   D3DXMESH_VB_SYSTEMMEM =      16 (&H10)
   D3DXMESH_VB_MANAGED  =      32 (&H20)
   D3DXMESH_VB_WRITEONLY =     64 (&H40)
   D3DXMESH_VB_DYNAMIC  =     128 (&H80)
   D3DXMESH_IB_SYSTEMMEM =    256 (&H100)
   D3DXMESH_SYSTEMMEM   =     272 (&H110)
   D3DXMESH_IB_MANAGED  =     512 (&H200)
   D3DXMESH_MANAGED     =     544 (&H220)
   D3DXMESH_WRITEONLY   =     816 (&H330)
   D3DXMESH_IB_WRITEONLY =   1024 (&H400)
```

IDH_CONST_D3DXERR_graphicsd3dxvb
IDH_CONST_D3DXMESH_graphicsd3dxvb

```
    D3DXMESH_IB_DYNAMIC   =     2048 (&H800)
    D3DXMESH_DYNAMIC      =     2176 (&H880)
    D3DXMESH_VB_SHARE     =     4096 (&H1000)
    D3DXMESH_USEHWONLY    =     8192 (&H2000)
    D3DDEVCAPS_NPATCHES   = 16777216 (&H1000000)
End Enum
```

Constants

D3DXMESH_32BIT
: The mesh has 32-bit indices instead of 16-bit indices. A 32-bit mesh can support up to 2^32-1 faces and vertices. This constant is currently not supported and should not be used.

D3DXMESH_DONOTCLIP
: Use the D3DUSAGE_DONOTCLIP usage flag for vertex and index buffers.

D3DXMESH_POINTS
: Use the D3DUSAGE_POINTS usage flag for vertex and index buffers.

D3DXMESH_RTPATCHES
: Use the D3DUSAGE_RTPATCHES usage flag for vertex and index buffers.

D3DXMESH_VB_SYSTEMMEM
: Use the D3DPOOL_SYSTEMMEM memory class for vertex buffers.

D3DXMESH_VB_MANAGED
: Use the D3DPOOL_MANAGED memory class for vertex buffers.

D3DXMESH_VB_WRITEONLY
: Use the D3DUSAGE_WRITEONLY usage flag for vertex buffers.

D3DXMESH_VB_DYNAMIC
: Use the D3DUSAGE_DYNAMIC usage flag for vertex buffers.

D3DXMESH_DYNAMIC
: Equivalent to specifying both D3DXMESH_VB_DYNAMIC and D3DXMESH_IB_DYNAMIC.

D3DXMESH_IB_SYSTEMMEM
: Use the D3DPOOL_SYSTEMMEM memory class for index buffers.

D3DXMESH_SYSTEMMEM
: Equivalent to specifying both D3DXMESH_VB_SYSTEMMEM and D3DXMESH_IB_SYSTEMMEM.

D3DXMESH_IB_MANAGED
: Use the D3DPOOL_MANAGED memory class for index buffers.

D3DXMESH_MANAGED
: Equivalent to specifying both D3DXMESH_VB_MANAGED and D3DXMESH_IB_MANAGED.

D3DXMESH_WRITEONLY
: Equivalent to specifying both D3DXMESH_VB_WRITEONLY and D3DXMESH_IB_WRITEONLY.

D3DXMESH_IB_WRITEONLY
Use the D3DUSAGE_WRITEONLY usage flag for index buffers.

D3DXMESH_IB_DYNAMIC
Use the D3DUSAGE_DYNAMIC usage flag for index buffers.

D3DXMESH_VB_SHARE
Forces the cloned meshes to share vertex buffers. These meshes will have separate index buffers that index into the shared vertex buffer.

D3DXMESH_USEHWONLY
Use hardware processing only. This flag should only be specified for a hardware processing device. On a mixed mode device this flag will cause the system to either use hardware only, or if the hardware is not capable it will approximate using the software capabilities.

Note that this flag is only valid when specified as an option for the **D3DXSkinMesh.ConvertToBlendedMesh** method.

D3DDEVCAPS_NPATCHES
Specifying this flag causes the vertex and index buffer of the mesh to be created with the D3DUSAGE_NPATCHES flag. This is required if the mesh object is to be rendered using N-Patch enhancement using Microsoft® Direct3D®.

See Also

D3DX8.CreateMesh, **D3DX8.CreateMeshFVF**, **D3DXBaseMesh.CloneMesh**, **D3DXBaseMesh.CloneMeshFVF**, **D3DXBaseMesh.GetOptions**, **D3DXPMesh.ClonePMesh**, **D3DXPMesh.ClonePMeshFVF,** **D3DXSPMesh.CloneMesh**, **D3DXSPMesh.CloneMeshFVF**, **D3DXSPMesh.ClonePMesh**, **D3DXSPMesh.ClonePMeshFVF**, **D3DXSPMesh.GetOptions**

CONST_D3DXMESHMISC

#Defines miscellaneous flags.

```
Enum CONST_D3DXMESHMISC
   UNUSED16        = 65535 (&HFFFF)
   UNUSED32        =    -1 (&HFFFFFFFF)
   MAX_FVF_DECL_SIZE =    15
End Enum
```

Constants

UNUSED16
This value is not used.

UNUSED32

IDH_CONST_D3DXMESHMISC_graphicsd3dxvb

This value is not used

MAX_FVF_DECL_SIZE
The upper limit of a declarator array.

See Also

D3DX8.LoadMeshFromX, **D3DX8.LoadMeshFromXof**, **D3DXMesh.Optimize**

CONST_D3DXMESHOPT

#Defines the type of optimizations that can be performed on a mesh.

```
Enum CONST_D3DXMESHOPT
   D3DXMESHOPT_COMPACT     = 1
   D3DXMESHOPT_ATTRSORT    = 2
   D3DXMESHOPT_VERTEXCACHE = 4
   D3DXMESHOPT_IGNOREVERTS = 16
   D3DXMESHOPT_SHAREVB     = 32
End Enum
```

Constants

D3DXMESHOPT_COMPACT
Reorders faces to remove unused vertices and faces.

D3DXMESHOPT_ATTRSORT
Reorders faces to optimize for fewer attribute bundle state changes and enhanced **D3DXBaseMesh.DrawSubset** performance.

D3DXMESHOPT_VERTEXCACHE
Reorders faces to increase the cache hit rate of vertex caches.

D3DXMESHOPT_IGNOREVERTS
Optimize the faces only; do not optimize the vertices.

D3DXMESHOPT_SHAREVB
Share vertex buffers.

See Also

D3DX8.LoadMeshFromX, **D3DX8.LoadMeshFromXof**, **D3DXMesh.Optimize**

CONST_D3DXMESHSIMP

#Defines simplification options for a mesh.

IDH_CONST_D3DXMESHOPT_graphicsd3dxvb
IDH_CONST_D3DXMESHSIMP_graphicsd3dxvb

```
Enum CONST_D3DXMESHSIMP
   D3DXMESHSIMP_VERTEX = 1
   D3DXMESHSIMP_FACE   = 2
End Enum
```

Constants

D3DXMESHSIMP_VERTEX
The mesh will be simplified by a specified number of vertices.

D3DXMESHSIMP_FACE
The mesh will be simplified by a specified number of faces.

See Also

D3DX8.GeneratePMesh, **D3DX8.SimplifyMesh**

CONST_DTFLAGS

#Defines flags used for drawing text.

```
Enum CONST_DTFLAGS
   DT_LEFT              =      0
   DT_TOP               =      0
   DT_CENTER            =      1
   DT_RIGHT             =      2
   DT_VCENTER           =      4
   DT_BOTTOM            =      8
   DT_WORDBREAK         =     16 (&H10)
   DT_SINGLELINE        =     32 (&H20)
   DT_EXPANDTABS        =     64 (&H40)
   DT_TABSTOP           =    128 (&H80)
   DT_NOCLIP            =    256 (&H100)
   DT_EXTERNALLEADING   =    512 (&H200)
   DT_CALCRECT          =   1024 (&H400)
   DT_NOPREFIX          =   2048 (&H800)
   DT_INTERNAL          =   4096 (&H1000)
   DT_EDITCONTROL       =   8192 (&H2000)
   DT_PATH_ELLIPSIS     =  16384 (&H4000)
   DT_END_ELLIPSIS      =  32768 (&H8000)
   DT_MODIFYSTRING      =  65536 (&H10000)
   DT_RTLREADING        = 131072 (&H20000)
   DT_WORD_ELLIPSIS     = 262144 (&H40000)
   DT_NOFULLWIDTHCHARBREAK = 524288 (&H80000)
   DT_HIDEPREFIX        = 1048576 (&H100000)
```

IDH_CONST_DTFLAGS_graphicsd3dxvb

```
DT_PREFIXONLY        = 2097152 (&H200000)End Enum
```

Constants

DT_LEFT
Aligns text to the left.

DT_TOP
Top-justifies text (single line only).

DT_CENTER
Centers text horizontally in the rectangle.

DT_RIGHT
Aligns text to the right.

DT_VCENTER
Centers text vertically (single line only).

DT_BOTTOM
Justifies text to the bottom of the rectangle. This value must be combined with DT_SINGLELINE.

DT_WORDBREAK
Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the *DestRect* parameter. A carriage return/line feed sequence also breaks the line.

DT_SINGLELINE
Displays text on a single line only. Carriage returns and line feeds do not break the line.

DT_EXPANDTABS
Expands tab characters. The default number of characters per tab is eight. The DT_WORD_ELLIPSIS, DT_PATH_ELLIPSIS, and DT_END_ELLIPSIS values cannot be used with the DT_EXPANDTABS value.

DT_TABSTOP
Sets tab stops. Bits 15–8 (high-order byte of the low-order word) of the *Format* parameter specify the number of characters for each tab. The default number of characters per tab is eight. The DT_CALCRECT, DT_EXTERNALLEADING, DT_INTERNAL, DT_NOCLIP, and DT_NOPREFIX values cannot be used with the DT_TABSTOP value.

DT_NOCLIP
Draws without clipping. **DrawTextW** is somewhat faster when DT_NOCLIP is used.

DT_EXTERNALLEADING
Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.

DT_CALCRECT
Determines the width and height of the rectangle. If there are multiple lines of text, **DrawTextW** uses the width of the rectangle pointed to by the *SrcRect* parameter and extends the base of the rectangle to bound the last line of text.

If there is only one line of text, **DrawTextW** modifies the right side of the rectangle so that it bounds the last character in the line. In either case, **DrawTextW** returns the height of the formatted text but does not draw the text.

DT_NOPREFIX
Turns off processing of prefix characters. Normally, **DrawTextW** interprets the mnemonic-prefix character ampersand (&) as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off. Compare with DT_HIDEPREFIX and DT_PREFIXONLY.

DT_INTERNAL
Uses the system font to calculate text metrics.

DT_EDITCONTROL
Duplicates the text-displaying characteristics of a multiline edit control. Specifically, the average character width is calculated in the same manner as for an edit control, and the function does not display a partially visible last line.

DT_END_ELLIPSIS or DT_PATH_ELLIPSIS
Truncates the string without adding ellipses so that the result fits in the specified rectangle. The string is not modified unless the DT_MODIFYSTRING flag is specified.

Specify DT_END_ELLIPSIS to truncate characters at the end of the string, or DT_PATH_ELLIPSIS to truncate characters in the middle of the string. If the string contains backslash (\) characters, DT_PATH_ELLIPSIS preserves as much of the text as possible after the last backslash.

DT_MODIFYSTRING
Modifies the string to match the displayed text. This flag has no effect unless the DT_END_ELLIPSIS or DT_PATH_ELLIPSIS flag is specified.

DT_RTLREADING
Displays text in right-to-left reading order for bi-directional text when a Hebrew or Arabic font is selected. The default reading order for all text is left-to-right.

DT_WORD_ELLIPSIS
Truncates text that does not fit in the rectangle and adds ellipses.

DT_NOFULLWIDTHCHARBREAK
Microsoft® Windows® 98, Windows 2000: Prevents a line break at a DBCS (double-wide character string), so that the line breaking rule is equivalent to an SBCS (single-wide character string)strings. For example, this can be used in Korean windows to increase the readability of icon labels. This is only effective if DT_WORDBREAK is specified.

DT_HIDEPREFIX
Windows 2000: Ignores the ampersand (&) prefix character in the text. The letter that follows is not underlined, but other mnemonic-prefix characters are still processed.

Example:

input string: "A&bc&&d"

normal: "Abc&d"

HIDEPREFIX: "Abc&d"

Compare with DT_NOPREFIX and DT_PREFIXONLY.

DT_PREFIXONLY
Windows 2000: Draws only an underline at the position of the character following the ampersand (&) prefix character. Does not draw any character in the string.

Example:

input string: "A&bc&&d"

normal: "Abc&d"

PREFIXONLY: " "

Compare with DT_NOPREFIX and DT_HIDEPREFIX.

See Also

D3DXFont.DrawTextW, **D3DX8.DrawText**

Error Codes

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by methods included with the Direct3DX utility library. See the individual method descriptions for lists of the values that each can return.

D3DXERR_CANNOTATTRSORT
Attribute sort (D3DXMESHOPT_ATTRSORT) is not supported as an optimization technique.

D3DXERR_CANNOTMODIFYINDEXBUFFER
The index buffer cannot be modified.

D3DXERR_INVALIDDATA
The data is invalid.

D3DXERR_INVALIDMESH
The mesh is invalid.

D3DXERR_SKINNINGNOTSUPPORTED
Skinning is not supported.

D3DXERR_TOOMANYINFLUENCES
Too many influences specified.