

Direct3D Immediate Mode

Direct3D® Immediate Mode application programming interface (API) is part of the three-dimensional (3-D) graphics component of DirectX®. The information on Direct3D Immediate Mode is organized into the following topics:

- About Direct3D Immediate Mode
- Why Use Direct3D Immediate Mode?
- Getting Started with Immediate Mode
- Direct3D Immediate Mode Architecture
- Direct3D Immediate Mode Essentials
- Direct3D Immediate Mode Tutorials
- Direct3D Immediate Mode Reference
- Direct3D Immediate Mode Tools and Samples

About Direct3D Immediate Mode

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

Direct3D has two modes: Immediate Mode and Retained Mode. Retained Mode is a high-level 3-D API for programmers who require rapid development or who want the help of the Retained Mode built-in support for hierarchies and animation.

Direct3D Immediate Mode 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. Immediate Mode is a device-independent way for applications to communicate with accelerator hardware at a low level. Direct3D Retained Mode is built on top of Immediate Mode.

These are some of the advanced features of Direct3D:

- 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, and Windows 2000

- Support for the Intel MMX architecture

Developers who use Immediate Mode instead of Retained Mode are typically experienced in high-performance programming issues and may also be experienced in 3-D graphics. Your best source of information about Immediate Mode is the sample code included with this SDK; it illustrates how to put Direct3D Immediate Mode to work in real-world applications.

Why Use Direct3D Immediate Mode?

The world management of Immediate Mode is based on vertices, polygons, and commands that control them. It allows 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. Developers with existing 3-D applications and developers who need to achieve maximum performance by maintaining the thinnest possible layer between their application and the hardware should use Immediate Mode, instead of Retained Mode.

Direct3D Immediate Mode provides simple and straightforward methods to set up and render a 3-D scene. The key set of rendering methods are 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 DrawPrimitive Methods.

Immediate Mode allows a low-overhead connection to 3-D hardware. This low-overhead connection comes at a price; 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 what its capabilities are.

Getting Started with Immediate Mode

The following sections describe some of the technical concepts that you need to understand before you write programs that incorporate 3-D graphics. This is designed as a general overview. For information about specific topics, see Direct3D Immediate Mode Architecture and Direct3D Immediate Mode Essentials.

If you are already experienced in producing 3-D graphics, you can skim the following sections for information that is unique to Direct3D.

This section is divided into the following topics:

- 3-D Coordinate Systems and Geometry
- Matrices and Transformations

3-D Coordinate Systems and Geometry

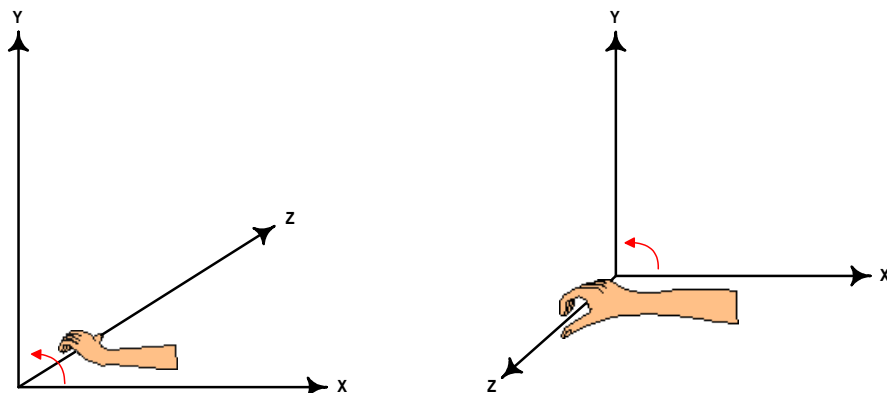
Programming Direct3D applications requires a working familiarity with 3-D geometric principles. This section introduces the most important geometric concepts for creating 3-D scenes. The following topics are covered:

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

The discussions in these topics provide you with a simple, 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 the positive z-axis points for that coordinate system.



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 v_0 , v_1 , v_2 , pass them to Direct3D as v_0 , v_2 , v_1 .

- 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. (Likewise, Visual Basic applications can perform this operation on the corresponding members of the **D3DMATRIX** type.)

There are a wide variety of other coordinate systems used in 3-D software. Left- and right-handed coordinates are the most common. However, it is not unusual for 3-D modeling programs 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 of 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.

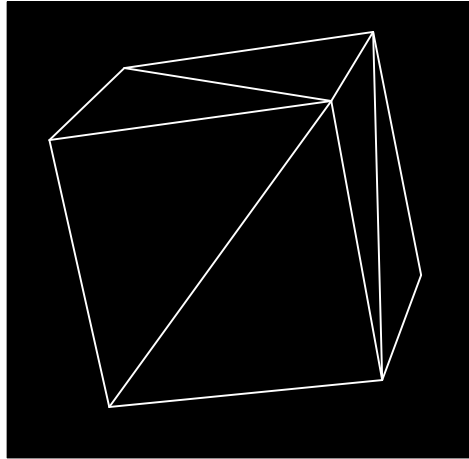
When you combine these operations, the results are not commutative—the order in which you multiply matrices is important.

3-D Primitives

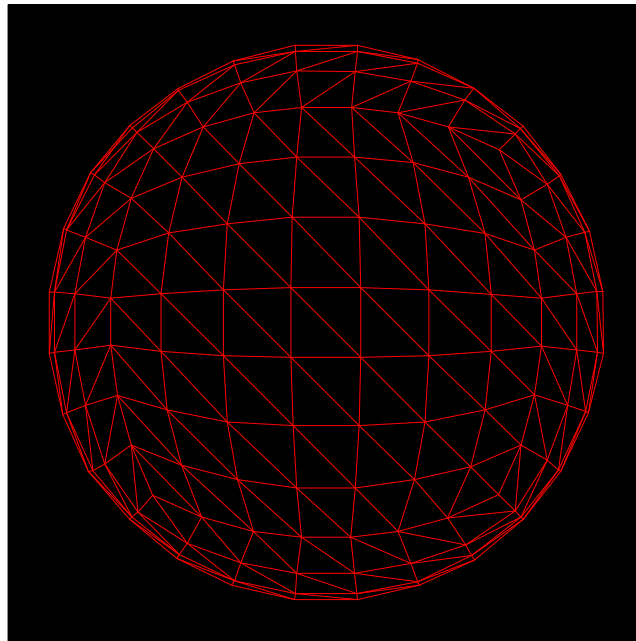
A 3-D primitive is a collection of vertices that form a single 3-D entity. The simplest primitive is a collection of points in a 3-D coordinate system, which is called a point list.

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. Direct3D uses triangles to compose most of its polygons because all three of the vertices in a triangle are guaranteed to be coplanar. Rendering non-planar 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 taken together 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, it looks curved when it is rendered. This is especially true if you use Gouraud shading. For details, see [Gouraud Shading](#).



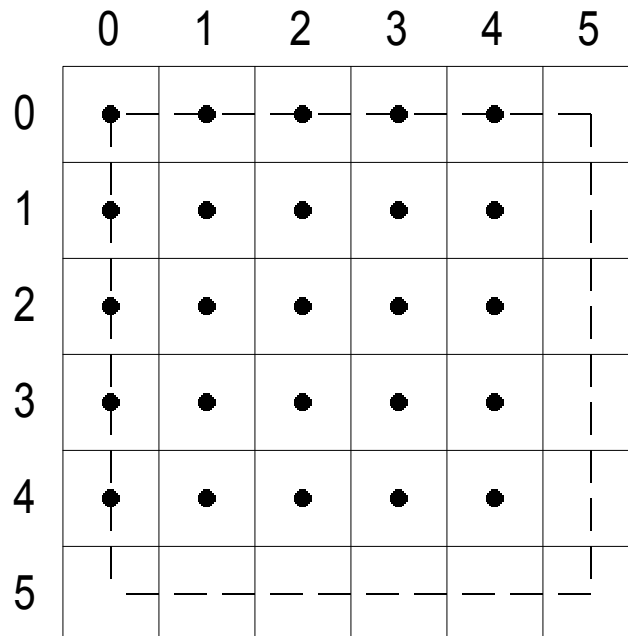
Triangle Rasterization Rules

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

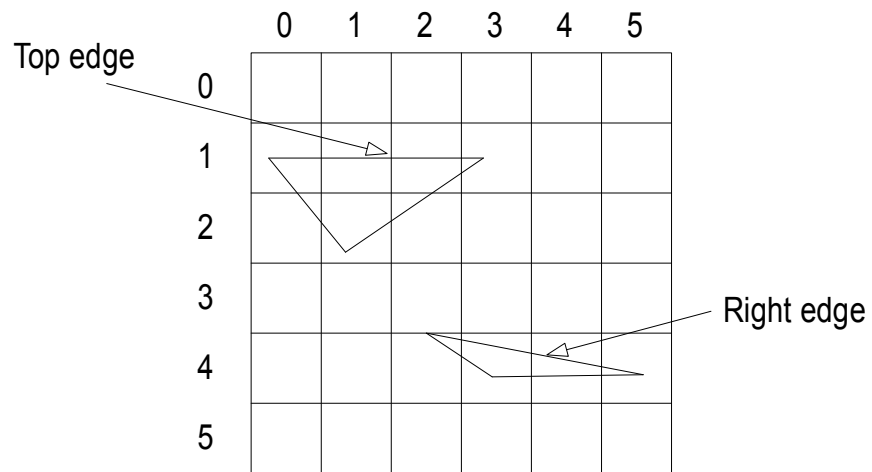
Direct3D uses a top-left filling convention for filling geometry. This is the same convention that is used for rectangles in GDI and OpenGL. In Direct3D, the center of the pixel is the decisive point. If the center is inside a triangle, the pixel is part of the triangle. Pixel centers are at integer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

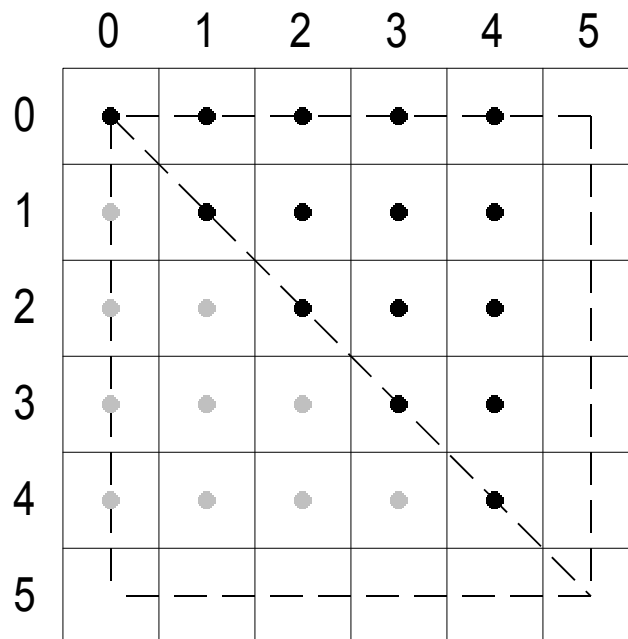
The following illustration shows a rectangle whose upper-left corner is at (0, 0) and whose lower-right corner is at (5, 5). This rectangle fills 25 pixels, just as you would expect. The width of the rectangle is defined as right minus left. The height is defined as bottom minus top.



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.

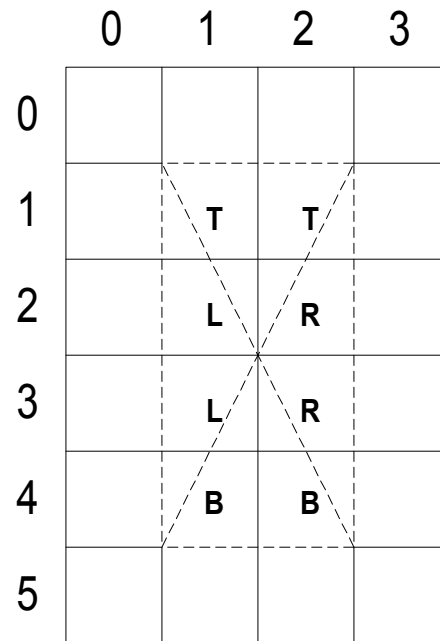


The top-left filling convention determines the action taken by Direct3D when a triangle passes through the center of a pixel. The following illustration shows two triangles, one at (0, 0), (5, 0), and (5, 5), and the other at (0, 5), (0, 0), and (5, 5). The first triangle in this case gets 15 pixels, whereas the second gets only 10 because the shared edge is the left edge of the first triangle.

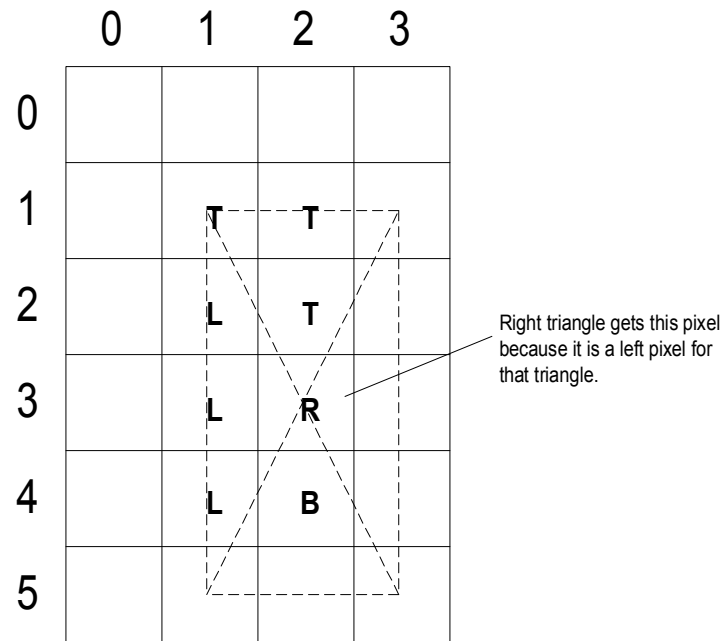


If you define a rectangle with its upper-left corner at (0.5, 0.5) and its lower-right corner at (2.5, 4.5), the center point of this rectangle is at (1.5, 2.5). When the Direct3D rasterizer tessellates this rectangle, the center of each pixel is unambiguously inside each of the four triangles, and the 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.

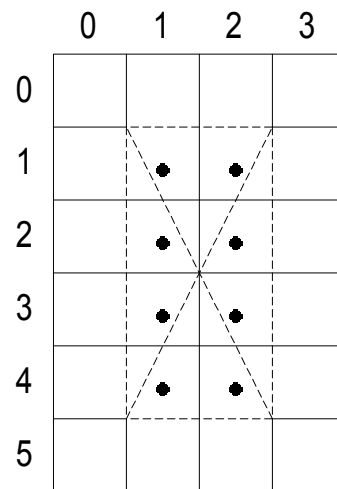


If you move the rectangle in the previous example so that its upper-left corner is at (1.0, 1.0), its lower-right corner at (3.0, 5.0), and its center point at (2.0, 3.0), Direct3D applies the top-left filling convention. Most of the pixels in this rectangle straddles the border between two or more triangles,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

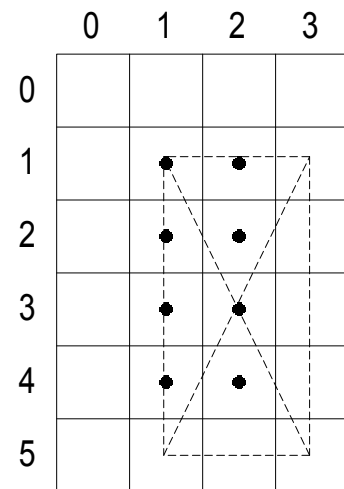


For both rectangles, the same pixels are affected.

(0.5, 0.5)-(2.5, 4.5)



(1.0, 1.0)-(3.0, 5.0)



Shading

This section describes techniques used in 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. Direct3D currently 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 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 (See Face and Vertex Normal Vectors). 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, utilizing the Gouraud shading mode and the RGB color model, the Direct3D lighting module would assign a red component of 0.6 to the pixel at the midpoint of the line between these vertices.

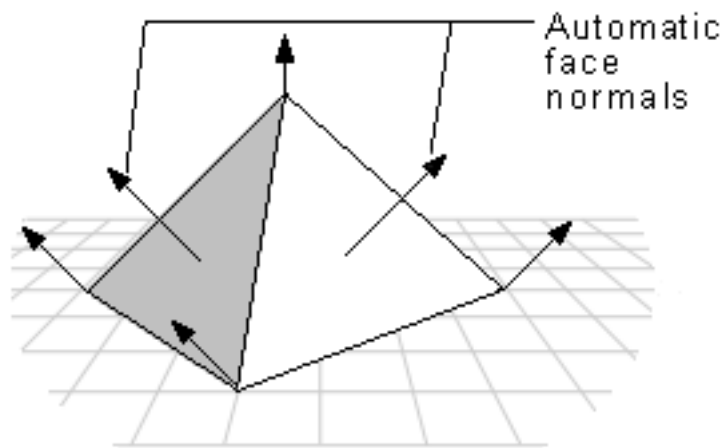
The following figure demonstrates Gouraud shading. This teapot is composed of many flat, triangular polygons. However, Gouraud shading makes the surface of the object appear curved and smooth.



Gouraud shading can also be used to display objects with sharp edges. For details, see Face and Vertex Normal Vectors.

Comparing Shading Modes

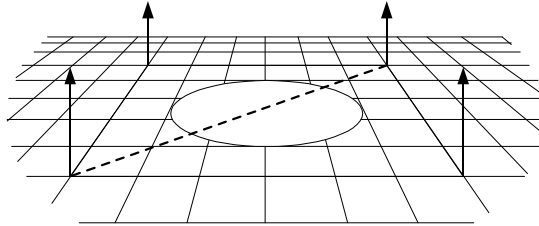
In the flat shading mode, the following pyramid would be displayed with a sharp edge between adjoining faces. In the Gouraud shading mode, however, shading values would be interpolated across the edge, and the final appearance would be of a curved surface.



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

Gouraud shading smooths out 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++]

Direct3D allows one shading mode to be selected at a time. By default, Gouraud shading is selected. In C++, the shading mode can be changed by calling the **IDirect3DDevice7::SetRenderState** method. Set the *dwRenderStateType* parameter to `D3DRENDERSTATE_SHADEMODE`. The *dwRenderState* parameter must be set to a member of the **D3DSHADEMODE** enumeration. The following sample code fragments 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 fragment assumes that lpDev is a valid pointer to
// an IDirect3DDevice7 interface.
hr = lpDev->SetRenderState(D3DRENDERSTATE_SHADEMODE, D3DSHADE_FLAT);
if(FAILED(hr))
{
    // Code to handle the error goes here.
}

// Set to Gouraud shading (this is the default for Direct3D).
hr = lpDev->SetRenderState(D3DRENDERSTATE_SHADEMODE,
    D3DSHADE_GOURAUD);
if(FAILED(hr))
{
    // Code to handle the error goes here.
}
```

[Visual Basic]

Direct3D allows one shading mode to be selected at a time. By default, Gouraud shading is selected. In Visual Basic, the shading mode can be changed by calling the **Direct3DDevice7.SetRenderState** method. Set the *state* parameter to

D3DRENDERSTATE_SHADEMODE. The *renderstate* parameter must be set to a member of the **CONST_D3DSHADEMODE** enumeration. The following code fragment 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 fragment assumes that d3dDev is a valid reference to
' a Direct3DDevice7 object.
On Local Error Resume Next
Call d3dDev.SetRenderState(D3DRENDERSTATE_SHADEMODE, _
                          D3DSHADE_FLAT)

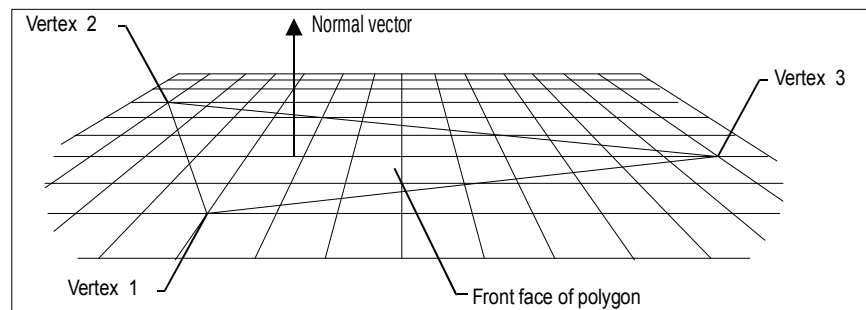
' Check for an error.
If Err.Number <> DD_OK Then
    ' Handle the error.
End If

' Set to Gouraud shading (this is the default for Direct3D).
Call d3dDev.SetRenderState(D3DRENDERSTATE_SHADEMODE, _
                          D3DSHADE_GOURAUD)

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

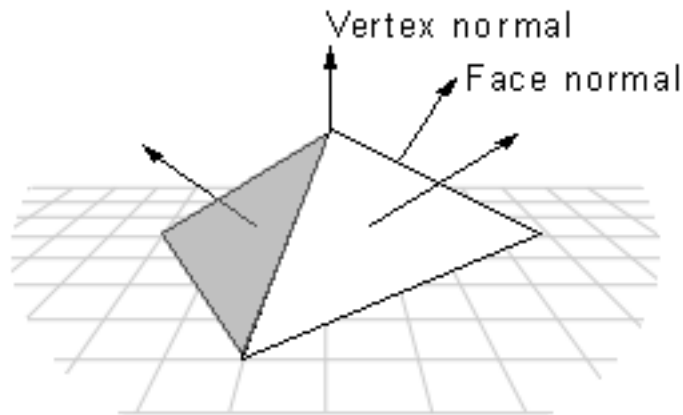
Face and Vertex Normal Vectors

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 Direct3D, only the front side 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.



[C++]

Direct3D applications written in C++ typically use the **D3DVERTEX** structure for their vertices. The members of the **D3DVERTEX** structure describe the position and orientation of the vertex. The orientation is indicated by a vertex normal vector. The following code fragment demonstrates how vertex values, including the vertex normal, can be set. The normal vectors point toward the viewport, which is at the origin of the world coordinate system. The vertex positions in this example are specified in world coordinates.

```
D3DVERTEX lpVertices[3];

// A vertex can be specified one structure member at a time.
lpVertices[0].x = 0;
lpVertices[0].y = 5;
lpVertices[0].z = 5;
lpVertices[0].nx = 0; // X component of the normal vector.
lpVertices[0].ny = 0; // Y component of the normal vector.
lpVertices[0].nz = -1; // Points the normal back at the origin.
lpVertices[0].tu = 0; // Only used if a texture is being used.
lpVertices[0].tv = 0; // Only used if a texture is being used.

// Vertices can also be specified on one line of code for each vertex
// by using some of the D3DOVERLOADS macros.
lpVertices[1] = D3DVERTEX(D3DVECTOR(-5,-5,5),D3DVECTOR(0,0,-1),0,0);
lpVertices[2] = D3DVERTEX(D3DVECTOR(5,-5,5),D3DVECTOR(0,0,-1),0,0);
```

[Visual Basic]

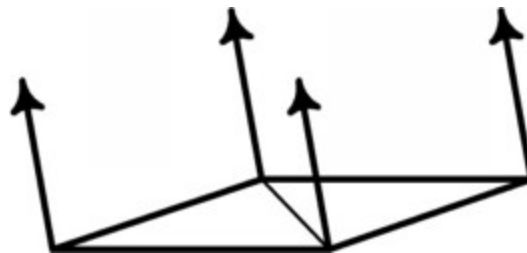
Direct3D applications written in Visual Basic typically use the **D3DVERTEX** type for their vertices. The members of the **D3DVERTEX** type describe the position and orientation of the vertex. The orientation is indicated by a vertex normal vector. The following code fragment demonstrates how vertex values, including the vertex normal, can be set. The normal vectors point toward the viewport, which is at the origin of the world coordinate system. The vertex positions in this example are specified in world coordinates.

```
D3DVERTEX Vertices(3)

' A vertex can be specified one structure member at a time.
Vertices(0).x = 0
Vertices(0).y = 5
Vertices(0).z = 5
Vertices(0).nx = 0 ' X component of the normal vector.
Vertices(0).ny = 0 ' Y component of the normal vector.
Vertices(0).nz = -1 ' Points the normal back at the origin.
Vertices(0).tu = 0 ' Only used if a texture is being used.
Vertices(0).tv = 0 ' Only used if a texture is being used.
```

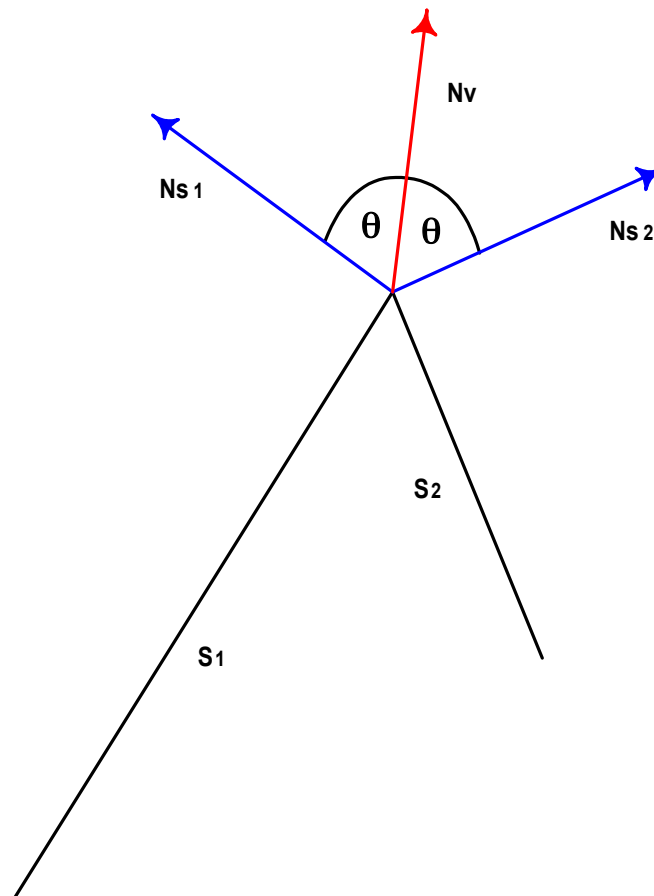
When applying Gouraud shading to a polygon, Direct3D uses the vertex normals to calculate the angle between the light source and the surface. It calculates the color and intensity values for the vertices and interpolates them for every point across all of the primitive's surfaces. Direct3D calculates 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.

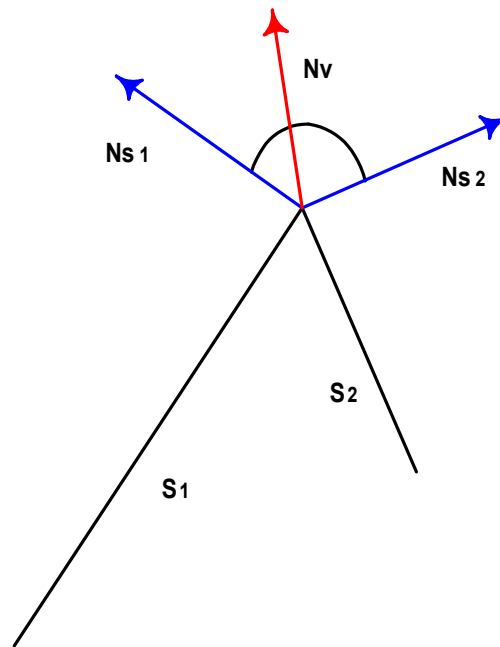


It is more likely, however, that your object is made up of triangle strips and the triangles are not coplanar. One simple way to get smooth shading across all of the triangles in the strip is to first calculate the surface normal vector for each polygonal face with which the vertex is associated. The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

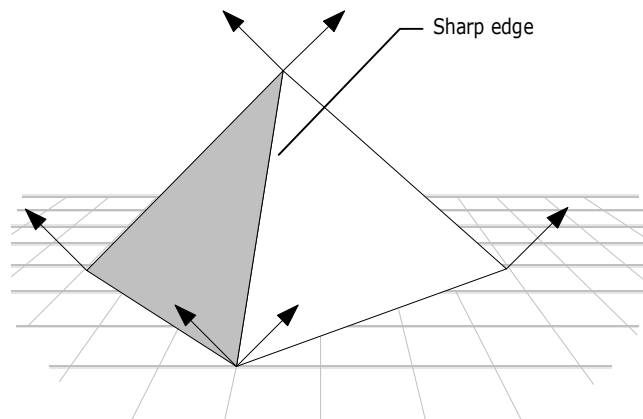
This method is illustrated by the following figure, which shows two surfaces, S1 and S2 seen edge-on from above. The normal vectors for S1 and S2 are shown in blue. The vertex normal vector is shown in red. The angle that the vertex normal vector makes with the surface normal of S1 is the same as the 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 it would if the vertex normal had equal angles with the surface normals.



You can use Gouraud shading to display some of the 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.



[C++]

If you are using the **IDirect3DDevice7::DrawPrimitive** or **IDirect3DDevice7::DrawIndexedPrimitive** 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. Since 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.

[Visual Basic]

If you are using the **Direct3DDevice7.DrawPrimitive** or **Direct3DDevice7.DrawIndexedPrimitive** 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. Since 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:

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.

A C++ application uses the **dwShadeCaps** member of the **D3DPRIMCAPS** structure to determine what forms of interpolation the current device driver supports. Similarly, a Visual Basic application accesses this information through the **lShadeCaps** member of the **D3DPRIMCAPS** type.

Matrices and Transformations

Use matrices in 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 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

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

Matrices in Direct3D are represented by a 4×4 homogenous matrix, defined by the **D3DMATRIX** structure in C++, and by the **D3DMATRIX** type in Visual Basic.

[C++]

The D3D_OVERLOADS implementation of the **D3DMATRIX** structure (**D3DMATRIX (D3D_OVERLOADS)**) 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, C++ programmers can refer to them by row and column number and index these numbers as needed. These indices are zero-based, so, for example, the element in the third row, second column would be $M(2, 1)$. To use the D3D_OVERLOADS operators, define D3D_OVERLOADS before including the D3dtypes.h header file.

The D3dutil.cpp source file also provides helper functions for creating and concatenating matrices. You can use these functions as they are or as a basis to write your own matrix manipulation functions.

[C++,Visual Basic]

3-D Transformations

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 Immediate Mode, see Geometry Pipeline.

About 3-D Transformations

In programs that work with 3-D graphics, you can use geometrical transformations to:

- 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 used to reinterpret 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'):

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

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++, Direct3D Immediate Mode 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:

```
D3DMATRIX scale = {
    D3DVAL(s), 0,      0,      0,
    0,          D3DVAL(s), 0,      0,
    0,          0,      D3DVAL(s), 0,
    0,          0,      0,      D3DVAL(1)
};
```

[Visual Basic]

In Visual Basic, Direct3D Immediate Mode 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:

```
Dim ScaleMatrix As D3DMATRIX
```

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

```
With ScaleMatrix
```

```
    .rc11 = s
```

```
    .rc22 = s
```

```
    .rc33 = s
```

```
    .rc44 = 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:

```
D3DMATRIX Translate(const float dx, const float dy, const float dz)
{
    D3DMATRIX ret = D3DUtil_SetIdentityMatrix(); // declared in d3dutil.h
    ret(3, 0) = dx;
    ret(3, 1) = dy;
    ret(3, 2) = dz;
    return ret;
} // End of Translate
```

[Visual Basic]

In Visual Basic, you can create a translation matrix by hand, or you can use the **TranslateMatrix** helper subroutine in the Math.bas file that is included with this SDK. The following example shows the source code for the **TranslateMatrix** subroutine:

```
Sub TranslateMatrix(m As D3DMATRIX, v As D3DVECTOR)
    Call dx.IdentityMatrix(m) ' Method of the DirectX7 object
    m.rc41 = v.x
    m.rc42 = v.y
    m.rc43 = v.z
End Sub
```

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:

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

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 **D3DUtil_SetRotateXMatrix**, **D3DUtil_SetRotateYMatrix**, and **D3DUtil_SetRotateZMatrix** helper functions in the D3dutil.cpp file to create rotation matrices. (The D3dutil.cpp file is included with this SDK.) The following is the sample code for the **D3DUtil_SetRotateXMatrix** helper function:

```
VOID D3DUtil_SetRotateXMatrix( D3DMATRIX& mat, FLOAT fRads )
{
    D3DUtil_SetIdentityMatrix( mat );
    mat._22 = cosf( fRads );
    mat._23 = sinf( fRads );
    mat._32 = -sinf( fRads );
    mat._33 = cosf( fRads );
}
```

[\[Visual Basic\]](#)

Visual Basic applications can use the **DirectX7.RotateXMatrix**, **DirectX7.RotateYMatrix**, and **DirectX7.RotateZMatrix** methods to create rotation matrices.

If you manually created a matrix for rotation about an axis—in this case, the x-axis—the source code would look 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)

    Call dx.IdentityMatrix(ret) ' Method of the DirectX7 object.

    ret.rc22 = cosine
    ret.rc23 = sine
    ret.rc32 = -sine
    ret.rc33 = 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 of the primary advantages 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:

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

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++ applications can use the `D3dmath.cpp` source file that is included with the DirectX SDK. It contains the **D3DMath_MatrixMultiply** helper function to perform matrix multiplication. Visual Basic applications can use the **MatrixMult** subroutine from the `Math.bas` file that is included with this SDK.

Notice the order in which the matrix multiplication is performed—the order 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 the following example. Imagine that you are creating the world transformation matrix for a stereotypical flying saucer. You would probably want to spin the UFO around its center (the y-axis of model space) and translate it to some other place in your scene. To accomplish this effect, you would first create a rotation matrix, then multiply it by a translation matrix, as 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 would have the visual effect of translating the UFO to its world space position, 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.

Direct3D Immediate Mode Architecture

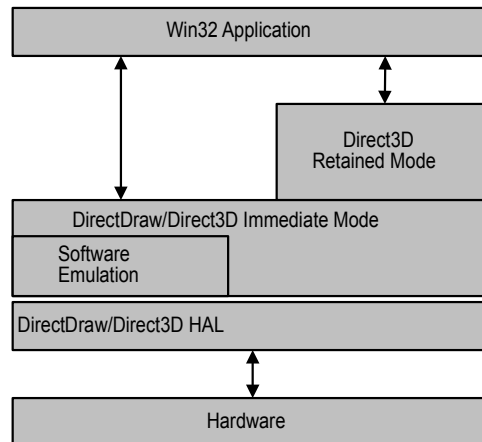
This section provides high-level information about the organization of the Direct3D® Immediate Mode documentation. Information is divided into the following topics:

- Architectural Overview of Immediate Mode
- Immediate Mode Object Types
- Immediate Mode COM Interfaces

Architectural Overview of Immediate Mode

Direct3D applications communicate with graphics hardware in a similar fashion whether they use Retained Mode or Immediate Mode. They may or may not take advantage of software emulation before interacting with the hardware emulation layer

(HAL). Since Direct3D is an interface to a DirectDraw® object, the HAL is referred to as the DirectDraw/Direct3D HAL.



Direct3D is tightly integrated with the DirectDraw component of DirectX. DirectDraw surfaces are used as rendering targets (front and back surfaces) and as z-buffers. The Direct3D interface is actually an interface to a DirectDraw object.

Immediate Mode Object Types

Direct3D Immediate Mode is made up of a series of objects. When programming with C++, you work directly with these objects to manipulate your virtual world and build a Direct3D application. A Visual Basic application ultimately accesses these objects as well, although DirectX for Visual Basic exposes a slightly different object model.

[C++]

DirectDraw Object

A DirectDraw object provides the functionality of Direct3D; **IDirect3D**, **IDirect3D2**, **IDirect3D3**, and **IDirect3D7** are interfaces to a DirectDraw object. Since a DirectDraw object represents the display device and the display device implements many of the most important features of Direct3D, the abilities of Direct3D are incorporated into DirectDraw. The **IDirect3D7** interface is an interface to an entirely different object than its predecessors. For more information, see Direct3D Interfaces. Like Direct3D, the method by which you create the DirectDraw object determines what interfaces you can use.

Create a DirectDraw object that can support the newest Direct3D interface by calling the **DirectDrawCreateEx** function. For more information, see Direct3D Interfaces.

DirectDrawSurface Object

A DirectDrawSurface object that was created with 3-D capabilities (DDSCAPS_3DDEVICE) can be used to create a Direct3D device object that uses the surface as its rendering target. Applications create a device from a surface by calling the **IDirect3D7::CreateDevice** method.

A **DirectDrawSurface** object that was created with texture-mapping capabilities contains the bitmap(s) that your Direct3D application uses for texturing objects in a scene. Applications use a pointer to the surface's **IDirectDrawSurface7** interface to identify the texture to Direct3D. Applications create a texture and retrieve its interface pointer by calling the **IDirectDraw7::CreateSurface** method. (This is a change from the method by which applications previously created textures, retrieving the **IDirect3DTexture2** interface for the surface by calling the **IUnknown::QueryInterface** method, and specifying the IID_IDirect3DTexture2 reference identifier.)

For more information, see Textures.

Direct3DDevice Object

A **Direct3DDevice** object encapsulates and stores the rendering state for an Immediate Mode application. Direct3D devices are independent of the surfaces they use as rendering targets, and devices can be used to render to several render-target surfaces (prior to DirectX 5.0, this was not the case). In DirectX 7.0, the recommended device object exposes the **IDirect3DDevice7** interface. Direct3D applications can create a device and retrieve a pointer to its **IDirect3DDevice7** interface by calling the **IDirect3D7::CreateDevice** method. A device object created in this manner only exposes the most recent interface. For details, see Device Interfaces and Direct3D Devices.

Unlike its predecessors, a device object that exposes the **IDirect3DDevice7** interface does not employ separate objects for materials, lights, and viewports; these objects are effectively obsolete with DirectX 7.0. Rather, the new interface includes methods that provides the functionality of these legacy objects as part of the internal data structures for the device.

Direct3DVertexBuffer Object

A **Direct3DVertexBuffer** object is a memory buffer that contains vertices to be rendered with the vertex-buffer rendering methods offered in the **IDirect3DDevice7** interface. Vertex buffers are not to be confused with the legacy execute buffer architecture. A vertex buffer contains only vertex data to be processed by a **Direct3DDevice** object and offers features to improve performance during rendering. To create a vertex buffer and retrieve an interface for it, call the **IDirect3D7::CreateVertexBuffer** method.

For more information, see Vertex Buffers.

[\[Visual Basic\]](#)

DirectX for Visual Basic exposes the following classes for Direct3D Immediate Mode programming:

DirectX7 Class

The **DirectX7** class defines the object that is used to create all other component-level objects, such as the **Direct3D7** or **DirectDraw7** object.

DirectDraw7 Class

The **DirectDraw7** class represents the display device, spawns the **Direct3D7** class, and creates the texture and rendering target surfaces used for rendering. To

create an object of the **DirectDraw7** class, call the **DirectX7.DirectDrawCreate** method.

DirectDrawSurface7 Class

The **DirectDrawSurface7** class represents memory that is used to contain image data for display, texturing, or as a rendering target for a Direct3D device. Call the **DirectDraw7.CreateSurface** method to create an empty surface, or call the **DirectDraw7.CreateSurfaceFromFile** method to create a surface and load it with image data from a bitmap file.

Direct3D7 Class

The **Direct3D7** class defines an object that provides the functionality of Direct3D, used to create and manipulate the objects that you need to render a scene. To create an object of the **Direct3D7** class, call the **DirectDraw7.GetDirect3D** method.

Direct3DDevice7 Class

The **Direct3DDevice7** class defines an object that encapsulates and stores the rendering state, lighting and material states, and viewport parameters for an Immediate Mode application written in Visual Basic. To create an object of this class, call the **Direct3D7.CreateDevice** method. For more information, see Direct3D Devices, Lighting and Materials, and Viewports and Clipping.

Direct3DEnumDevices Class

The **Direct3DEnumDevices** class defines an object that can enumerate the rendering devices present on a system. To create an object of the **Direct3DEnumDevices** class, call the **Direct3D7.GetDevicesEnum** method.

Direct3DEnumPixelFormat Class

The **Direct3DEnumPixelFormat** class defines an object that can enumerate the pixel formats supported for a given device on a system. To create an object of the **Direct3DEnumPixelFormat** class, call the **Direct3D7.GetEnumZBufferFormats** or **Direct3DDevice7.GetTextureFormatsEnum** methods.

Direct3DVertexBuffer7 Class

The **Direct3DVertexBuffer7** class defines an object that acts as a memory buffer for vertices. Vertices in a vertex buffer can be rendered with the vertex-buffer rendering methods offered in the **Direct3DDevice7** class. Call the **Direct3D7.CreateVertexBuffer** method to create a vertex buffer.

For more information, see Vertex Buffers.

Immediate Mode COM Interfaces

[\[Visual Basic\]](#)

Note

The information in this topic applies only to applications written in C++. DirectX for Visual Basic does not expose COM interfaces.

[C++]

The Direct3D Immediate Mode API consists primarily of the following interfaces:

IDirect3D7	Root interface, used to obtain other interfaces
IDirect3DDevice7	3-D Device for DrawPrimitive-based programming
IDirect3DVertexBuffer7	Interface used to work with vertex buffers.

For backward compatibility with previous versions of DirectX, all objects that supported former interface versions still support them in DirectX 7.0. However, new objects may only support the latest interfaces. For more information, see Direct3D Interfaces and Device Interfaces.

Direct3D Immediate Mode Essentials

Direct3D Immediate Mode consists of a relatively small number of API elements that create objects, fill them with data, and link them together. The API is based on COM. The Immediate Mode API is a very thin layer over the Direct3D drivers.

This section provides technical information about the components of Direct3D Immediate Mode. Information is divided into the following topics.

- Immediate Mode Changes for DirectX 7.0
- Direct3D and DirectDraw
- Direct3D Devices
- Geometry Pipeline
- Lighting and Materials
- Vertex Formats
- Textures
- Depth Buffers
- Stencil Buffers
- Vertex Buffers
- Common Techniques and Special Effects
- GUIDs
- Performance Optimization
- Troubleshooting

Immediate Mode Changes for DirectX 7.0

[Visual Basic]

Support for Visual Basic applications

Like the other components in DirectX 7.0, Direct3D Immediate Mode now offers its services to Visual Basic applications. This is made possible by a set of classes that marshal data from the Visual Basic application to the DirectX run time. In addition, the documentation provided with this SDK has been expanded to include the Visual Basic developer.

[C++]

DirectX 7.0 maintains backward compatibility by exposing and supporting objects and interfaces offered by previous releases of DirectX. However, two key Direct3D interfaces have been added as part of DirectX 7.0: **IDirect3D7** and **IDirect3DDevice7**. They are interfaces to new objects that are distinct from those in previous releases. These new objects provide several new features, performance improvements, and ease-of-use enhancements over their counterparts from previous versions of DirectX:

Hardware-accelerated transformation and lighting

Direct3D can now take advantage of 3-D accelerators to accelerate transformation and lighting operations in hardware. If compliant hardware is available, the **IDirect3D7::EnumDevices** method sets the **D3DDEVCAPS_HWTRANSFORMANDLIGHT** capability flag in the **dwDevCaps** member of the associated **D3DDEVICEDESC7** structure. For more information, see TnLHAL Device.

Environment mapping with cubic environment maps

Direct3D and DirectDraw now support a special type of texture map used in environment mapping, called a cubic environment map. Cubic environment mapping involves the use of a six-sided texture that can contain images to be applied to objects in a scene. Cubic environment maps provide realistic environment mapping in your applications, are easy to use, and even allow you to implement dynamic environment mapping.

API Changes

Direct3D for DirectX 7.0 moves lighting, material, and viewport parameters into the device interface. Therefore, separate objects to handle these parameters are no longer needed, nor are their interfaces. Several methods have been included in the **IDirect3DDevice7** interface to manipulate these parameters.

Vertex buffers can now be used with strided vertices, made possible by the addition of the **IDirect3DVertexBuffer7::ProcessVerticesStrided** method to the new **IDirect3DVertexBuffer7** interface.

Geometry Blending

Applications that use the Direct3D geometry pipeline can take advantage of new support for geometry blending. Geometry blending can be used to perform

"skinning" effects to increase the realism of segmented objects in a scene, especially characters. For more information, see Geometry Blending.

Device-state blocks

Applications that target the new **IDirect3DDevice7** interface can take advantage of its support for programmable sets of state changes, called state blocks. State blocks make it possible for a Direct3D application to record common sequences of device state changes into a construct that has a unique identifier, or block handle. Applications can use the block handle to execute previously recorded state blocks in a single method call. Besides minimizing the calls required to change device states, state blocks enable devices to cache precompiled sets of state changes for optimal execution, often resulting in improved performance. For details, see State Blocks.

Improved texture management

The Direct3D texture manager has been expanded to allow applications to prioritize managed textures. Direct3D uses texture priorities to determine which textures to keep in memory, and which to remove.

Support for Visual Basic applications

Like the other components in DirectX 7.0, Direct3D Immediate Mode now offers its services to Visual Basic applications. This is made possible by a set of classes that marshal data from the Visual Basic application to the DirectX run time. In addition, the documentation provided with this SDK has been expanded to include the Visual Basic developer.

Enhanced software emulation

Direct3D has been optimized to use any special instructions supported by the CPU. Supported instruction sets include the AMD 3D-Now! instruction set on some AMD processors and the MMX instruction set supported by many Intel processors. Where available, Direct3D utilizes the 3D-Now! instruction set to accelerate transformation and lighting operations and the MMX instruction set to accelerate rasterization. Applications that use the Direct3D transformation and lighting pipeline with software devices automatically benefit from this feature.

Migrating to the new set of Direct3D objects comes at a small price. Despite the names of the interfaces that they support, these objects are separate from their counterparts in previous releases, and they do not support the same interfaces. You cannot use **IUnknown::QueryInterface** to retrieve a legacy interface from an object that supports the new interfaces, nor can you query for a new interface from a legacy object. Attempts to do so fail, returning **E_NOINTERFACE**.

Existing applications that use execute buffers for rendering cannot use the new objects unless they abandon the execute buffer architecture. Otherwise, they must use the legacy objects (which will continue to be supported). In fact, execute-buffer rendering is generally considered obsolete. Therefore, no information covering the use of execute buffers is included in this documentation. Documentation about execute-buffer rendering is still available in the legacy documentation that ships with this SDK and is available for download from <http://www.microsoft.com/directx>.

Direct3D and DirectDraw

This section describes the close relation between Direct3D and DirectDraw. It offers information on the following topics:

- The DirectDraw Object and Direct3D
- Direct3D Interfaces
- Accessing Direct3D
- Creating Objects Subordinate to Direct3D
- DirectDraw Cooperative Levels and FPU Precision

The DirectDraw Object and Direct3D

Direct3D is implemented through COM objects and interfaces. Applications written in C++ directly access these interfaces and objects, whereas Visual Basic application interact with a layer of code—visible as the DirectX for Visual Basic Classes—that marshals data from a Visual Basic application to the DirectX run time. The information specific to C++ in this topic describes important relationships between DirectDraw and Direct3D. The discussion is followed by details about similar relationships between the classes exposed by DirectX for Visual Basic.

[C++]

The Direct3D interfaces are actually interfaces to the DirectDraw object. DirectDraw presents programmers with a single, unified object that encapsulates both the DirectDraw and Direct3D states. The DirectDraw object is the first object that your application creates and the last object that your application releases. Since the DirectDraw object represents the display device and the display device implements many of the most important features of Direct3D, the abilities of Direct3D are incorporated into DirectDraw.

Note

For DirectX 7.0, there are two ways to create the DirectDraw object from which you access Direct3D: the new **DirectDrawCreateEx** function and the legacy **DirectDrawCreate** function used in previous releases of DirectX. The **DirectDrawCreateEx** function creates a DirectDraw object that is capable of exposing the newest set of Direct3D interfaces offered in DirectX 7.0. The **DirectDrawCreate** function, on the other hand, creates a DirectDraw object that is incapable of spawning a Direct3D interface that supports the features of DirectX 7.0. This documentation assumes that all applications will create their DirectDraw objects by using the **DirectDrawCreateEx** function. For more information, see Direct3D Interfaces and Device Interfaces.

This means that the lifetime of the Direct3D driver state is the same as that of the DirectDraw object. Releasing the Direct3D interface does not destroy the Direct3D driver state. That state is not destroyed until all references to that object—whether they are DirectDraw or Direct3D references—have been released. Therefore, if you

release a Direct3D interface while holding a reference to a DirectDraw driver interface and then query the Direct3D interface again, the Direct3D state is preserved.

The DirectDraw object created by the **DirectDrawCreateEx** function contains a single Direct3D interface, **IDirect3D7**. (DirectDraw objects created with the legacy **DirectDrawCreate** function support the legacy interfaces **IDirect3D**, **IDirect3D2**, and **IDirect3D3**. These interfaces are obsolete and are provided for backward compatibility.) New applications should use the **IDirect3D7** interface to create other Direct3D objects. For details, see Direct3D Interfaces.

[\[Visual Basic\]](#)

Although the DirectDraw and Direct3D share much of the same code—in fact, they are basically the same root object—DirectX for Visual Basic exposes them according to how you will use them: as functionally separate classes. DirectX for Visual Basic uses the **DirectDraw7** and **Direct3D7** classes from which your application can create objects. The underlying relationship (as described in the C++ portion of this topic) between DirectDraw and Direct3D is apparent even from within Visual Basic. The global **DirectX7** class contains a method to create a **DirectDraw7** object, but no equivalent method for Direct3D. Rather, the **DirectDraw7** class defines a method to retrieve the **Direct3D7** class, called **DirectDraw7.GetDirect3D** because these two classes actually refer to the same object within DirectX.

The **GetDirect3D** method does not create a new object. Rather, the method creates a new class that exposes the Direct3D-related features offered by the DirectDraw object that created it.

The **DirectDraw7** class provides the two-dimensional features for a display device, whereas the **Direct3D7** class offers the 3-D features for the device. Use the **DirectDraw7** class to create the surfaces that the Direct3D class uses for rendering targets, depth buffers, and textures. The Direct3D class is then used to create the subordinate objects that Direct3D requires, such as vertex buffers and enumeration classes.

Direct3D Interfaces

[\[Visual Basic\]](#)

Note

The information in this topic applies only to applications written in C++. DirectX for Visual Basic does not directly expose COM interfaces to applications.

[\[C++\]](#)

The functionality of Direct3D objects is accessed by C++ applications through the interfaces **IDirect3D**, **IDirect3D2**, **IDirect3D3**, and **IDirect3D7**. This section

presents general information on all three of these interfaces, working backward from the **IDirect3D7** interface introduced with DirectX 7.0 to the first interface, **IDirect3D**.

IDirect3D7 Interface

The **IDirect3D7** interface, although named similarly to its predecessors, represents an interface to an entirely new Direct3D object. This object supports and expands upon the DrawPrimitive architecture, but it does not support the legacy interfaces mentioned previously in this topic. Applications retrieve a pointer to the **IDirect3D7** interface by calling the **QueryInterface** method of a DirectDraw object that was created by calling the **DirectDrawCreateEx** function.

The **IDirect3D7** interface supports many of the same features as the **IDirect3D3** interface, enabling applications to create rendering devices and vertex buffers, as well as enumerating pixel formats. This interface adds the ability to create textures, rendering obsolete the need to query a DirectDraw surface for a texture interface.

A notable difference between **IDirect3D7** and its ancestors is the absence of methods to create light, material, or viewport objects. This difference reflects the approach taken in the **IDirect3DDevice7** interface to incorporate lighting, material, and viewport parameters as part of the device's internal data structures, rather than separate objects.

The **IDirect3D7** object supports the features described in Immediate Mode Changes for DirectX 7.0.

Applications obtain a pointer to the **IDirect3D7** interface by using the **QueryInterface** method of the new **IDirectDraw7** interface. For details, see Accessing Direct3D.

IDirect3D2 and IDirect3D3 Interfaces

Applications get a pointer to the legacy **IDirect3D2** and **IDirect3D3** interfaces from a DirectDraw object that was created by the **DirectDrawCreate** function (as opposed to the new **DirectDrawCreateEx** function.) The interfaces are retrieved by calling the **QueryInterface** method for the DirectDraw object. Using these interface, your applications could find and enumerate the types of Direct3D devices supported by a particular DirectDraw object. They also exposed methods needed to create other Direct3D Immediate Mode objects, such as viewports, materials and lights. (The concept of lights, materials, and viewports as discrete objects is abandoned by the **IDirect3D7** interface.)

One of the most important differences between **IDirect3D2** and its predecessor, **IDirect3D**, is that **IDirect3D2** implements an **IDirect3D2::CreateDevice** method. This method creates a Direct3D device that supports the DrawPrimitive methods. For more information about the devices created by the **IDirect3D2::CreateDevice** method, see Direct3D Devices.

Like the **IDirect3D2** interface, the **IDirect3D3** interface supports the DrawPrimitive style of Direct3D programming. In addition, the **IDirect3D3** interface extends the functionality of DrawPrimitive-style programming by introducing the **IDirect3DDevice3** interface. This interface introduced the

concepts of flexible vertex formats, vertex buffers, and new texturing capabilities. For additional information, see Vertex Formats, Textures, Vertex Buffers and Rendering.

Applications obtain a pointer to the **IDirect3D3** interface from a DirectDraw object, using the **QueryInterface** method.

IDirect3D Interface

The legacy **IDirect3D** interface supports the use of execute buffers. It is provided for compatibility with existing code. New applications should use the **IDirect3D7** interface. If your application uses execute buffers, it cannot migrate to the features offered by DirectX 7.0. For more information, see Immediate Mode Changes for DirectX 7.0.

Accessing Direct3D

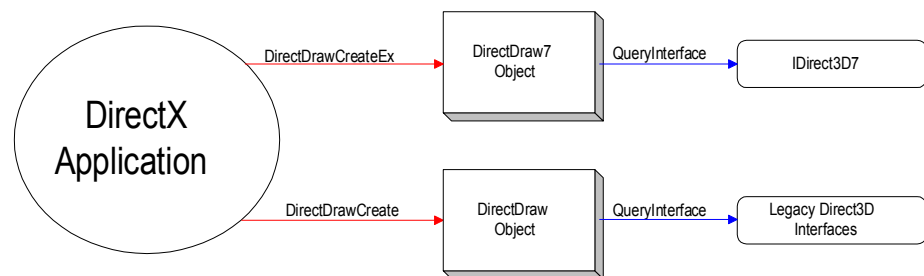
[C++]

When a Direct3D application written in C++ starts up, it must obtain a pointer to an **IDirect3D7** interface to access the latest Direct3D functionality. Use the following steps to obtain a pointer to the **IDirect3D7** interface:

0 To obtain a pointer to the IDirect3D7 interface

1. Call **DirectDrawCreateEx** to create a DirectDraw device. Applications must use the **DirectDrawCreateEx** function to create a DirectDraw object capable of supporting **IDirect3D7**; a DirectDraw object created by the legacy **DirectDrawCreate** function does not support the **IDirect3D7** interface.
2. Call the **IUnknown::QueryInterface** method to get a pointer to an **IDirect3D7** interface.

The following illustration shows these steps and relates this process to the one required for legacy applications:



Note

As shown in the preceding figure, the DirectDraw component is comprised of two COM objects. The newest object—called DirectDraw7 and created by calling the **DirectDrawCreateEx** function—is the only object that exposes the **IDirect3D7** interface. The DirectDraw7 object does not expose legacy Direct3D

interfaces. Applications that require previous iterations of the Direct3D must use the **DirectDrawCreate** method to create a DirectDraw object, and then query for a legacy interface.

The following code fragment demonstrates how to create a DirectDraw7 object and query for an **IDirect3D7** interface:

```
LPDIRECTDRAW7 lpDD;          // IDirectDraw7 Interface
LPDIRECT3D7   lpD3D;         // IDirect3D7 Interface
HRESULT hr;

// Get an IDirectDraw7 interface.
// Use the current display driver.
hr = DirectDrawCreateEx (NULL, (void **)&lpDD, IID_IDirectDraw7, NULL);
if (FAILED (hr))
{
    // Code to handle an error goes here.
}

// Get D3D interface
hr = lpDD->QueryInterface (IID_IDirect3D7, (void **)&lpD3D);
if (FAILED (hr))
{
    // Code to handle the error goes here.
}
```

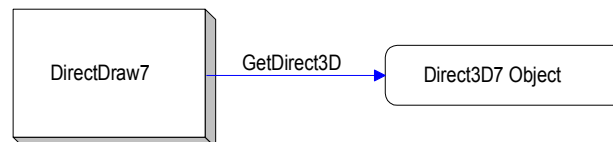
[Visual Basic]

When a Direct3D application written in Visual Basic starts up, it must obtain a reference to the **Direct3D7** class to access Direct3D functionality. Use the following steps to obtain such a reference:

0 To obtain a reference to the Direct3D7 class

1. Call **DirectX7.DirectDrawCreate** to create a **DirectDraw7** class object.
2. Call the **DirectDraw7.GetDirect3D** method to retrieve the DirectDraw object's reference to the **Direct3D7** class.

The following illustration shows these steps:



The following code fragment demonstrates how to retrieve the **Direct3D7** class from a **DirectDraw7** class object:

```
' Retrieve a reference to the Direct3D7 class.
```

```
'  
' For this example, dx is a valid reference to the DirectX7 class.  
On Local Error Resume Next  
Dim ddraw As DirectDraw7  
Dim d3d As Direct3D7  
  
' Create a default DirectDraw7 object.  
Set ddraw = dx.DirectDrawCreate("")  
If (Err.Number <> DD_OK) Then  
    ' Handle error.  
End If  
  
' Retrieve the Direct3D7 reference from the DirectDraw7 object.  
Set d3d = ddraw.GetDirect3D  
If (Err.Number <> DD_OK) Then  
    ' Handle error.  
End If
```

Creating Objects Subordinate to Direct3D

[C++]

Applications written in C++ use the **IDirect3D7** interface to create Direct3D rendering devices and vertex buffers. A Direct3D application creates a rendering device by invoking the **IDirect3D7::CreateDevice** method. For details, see Direct3D Devices. The rendering device encompasses the rendering, lighting, material, and viewport states.

Call the **IDirect3D7::CreateVertexBuffer** method to create a vertex buffer. Vertex buffers contain vertex data specially prepared for faster rendering than simple arrays of vertex data. For more information, see Vertex Buffers.

[Visual Basic]

Applications written in Visual Basic use the methods of the **Direct3D7** class to create Direct3D rendering devices and vertex buffers. A Direct3D application creates a rendering device by invoking the **Direct3D7.CreateDevice** method. For details, see Direct3D Devices. The rendering device encompasses the rendering, lighting, material, and viewport states.

Call the **Direct3D7.CreateVertexBuffer** method to create a vertex buffer. Vertex buffers contain vertex data specially prepared for faster rendering than simple arrays of vertex data. For more information, see Vertex Buffers.

DirectDraw Cooperative Levels and FPU Precision

[Visual Basic]

Note

The information in this topic applies only to applications written in C++. DirectX for Visual Basic does not support changing floating-point unit (FPU) precision.

[C++]

Direct3D always uses single-precision floating-point calculations to increase the performance of rendering a 3-D scene. By default, when you create a rendering device that supports the **IDirect3DDevice7** interface, Direct3D sets the FPU to single-precision, round-to-nearest mode with exceptions disabled. From that point on, the system assumes that the state of the FPU does not change until the device is released; it then resets the FPU to double-precision mode.

Note

This behavior is different from that of the legacy **IDirect3DDevice3** interface. Devices that expose **IDirect3DDevice3** check FPU mode on each rendering pass unless the `DDSCCL_FPUPRESERVE` flag was used when the DirectDraw cooperative level was set in a prior call to **IDirectDraw4::SetCooperative** method.

Applications that require double-precision accuracy can include the `DDSCCL_FPUPRESERVE` flag when calling the **IDirectDraw7::SetCooperativeLevel** method to set the DirectDraw cooperative level. When the `DDSCCL_FPUPRESERVE` flag is used, the system still uses single-precision mode, but it saves the state of the FPU prior to changing the precision mode, then restores the mode when it returns control to the application. (Calling the method again using the `DDSCCL_FPUPRESERVE` flag returns Direct3D to its default behavior.) Obviously, this check, save, then restore process takes time and can noticeably affect performance. This flag should only be used when the application must have double-precision accuracy and you do not want to include code to manually set and restore the FPU mode as the application requires.

Floating-point precision is thread-specific, so when developing multithreaded applications, be careful to check the FPU precision state to ensure that it is set and reset as appropriate for each thread.

Important

Loading some dynamic-link libraries (DLLs) at run time can cause the FPU to be reset to double-precision mode. Some compilers, such as Visual C++, set the default DLL entry point to **_DllMainCRTStartup**, a function that the compiler supplies to initialize the C/C++ run-time components. This function also sets the FPU precision mode to double precision. If an application sets the

DDSCL_FPUSETUP cooperative level, then loads a DLL, Direct3D does not detect that the FPU has been reset, and performance suffers.

If your application loads DLLs at run time, it should check and reset the FPU precision mode immediately after the **LoadLibrary** Win32 function returns, and before calling any Direct3D functions. You can reset the precision mode explicitly or by calling the **IDirectDraw7::SetCooperativeLevel** method again with the DDSCL_FPUSETUP flag set. Using **SetCooperativeLevel** to set the FPU precision mode can also cause DirectDraw surfaces to be lost.

You can explicitly set the entry point for DLLs that you compile by using the /ENTRY: linker switch. However, if you do, the C/C++ run time is not initialized automatically.

The following is a sample source file for a console application that checks and sets the FPU precision setting by using inline assembly language:

```
#include <windows.h>
#include <math.h>

// This function evaluates whether the floating-point
// control Word is set to single precision/round to nearest/
// exceptions disabled. If not, the
// function changes the control Word to set them and returns
// TRUE, putting the old control Word value in the passback
// location pointed to by pwOldCW.
BOOL MungeFPCW( WORD *pwOldCW )
{
    BOOL ret = FALSE;
    WORD wTemp, wSave;

    __asm fstcw wSave
    if (wSave & 0x300 ||      // Not single mode
        0x3f != (wSave & 0x3f) || // Exceptions enabled
        wSave & 0xC00)      // Not round to nearest mode
    {
        __asm
        {
            mov ax, wSave
            and ax, not 300h    ;; single mode
            or  ax, 3fh        ;; disable all exceptions
            and ax, not 0xC00   ;; round to nearest mode
            mov wTemp, ax
            fldcw wTemp
        }
        ret = TRUE;
    }
    *pwOldCW = wSave;
    return ret;
}
```

```
}

void RestoreFPCW(WORD wSave)
{
    __asm fldcw wSave
}

void __cdecl main()
{
    WORD wOldCW;
    BOOL bChangedFPCW = MungeFPCW( &wOldCW );
    // Do something with control Word, as set by MungeFPCW.
    if ( bChangedFPCW )
        RestoreFPCW( wOldCW );
}
```

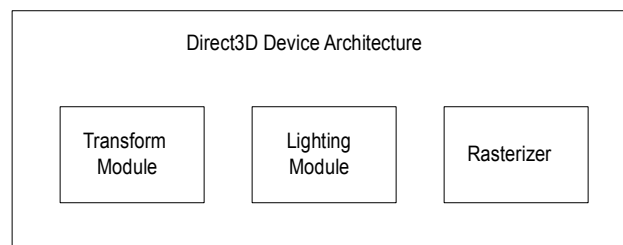
Direct3D Devices

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

- What Is a Direct3D Device?
- Direct3D Device Types
- Device Interfaces
- Device States
- Using Devices
- AGP Surfaces and Direct3D Devices

What Is a Direct3D Device?

A 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 DirectDraw 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 Formats](#).

[C++]

In C++, there are four COM interfaces that can be used for rendering:

IDirect3DDevice, **IDirect3DDevice2**, **IDirect3DDevice3**, and **IDirect3DDevice7**. In this documentation, it is assumed that applications target the latest interface version. For more information, see [Device Interfaces](#).

[Visual Basic]

In DirectX for Visual Basic, you access Direct3D Immediate Mode devices through the **Direct3DDevice7** class. The **Direct3DDevice7** class supports the `DrawPrimitive` family of scene-rendering methods. The **Direct3DDevice7** object can use various `DirectDraw` surfaces as render targets at different times if the application requires it. For more information, see **Direct3DDevice7.SetRenderTarget**.

Direct3D Device Types

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

- [About Device Types](#)
- [HAL Device](#)
- [TnLHAL Device](#)
- [RGB Device](#)
- [Reference Rasterizer](#)
- [Legacy Device Types](#)

About Device Types

Direct3D currently supports four types of Direct3D devices: the HAL device, a HAL device with accelerated transformation and lighting support, software-emulated RGB device, and the reference rasterizer. The first two types of devices can be used for shipping applications, and the reference rasterizer is supported for feature testing.

[C++]

Note

Previous releases of DirectX exposed additional device types—the MMX and Ramp devices—that are now obsolete. These are still available to C++ applications that target older versions of Direct3D but are not supported through the latest interfaces. For more information, see [Legacy Device Types](#).

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 better performance than software-emulated devices. In most cases, applications target computers that have hardware acceleration of some kind and fall back on software emulation to accommodate lower-end computers.

With the exception of the reference rasterizer, software devices do not always support the same features as a hardware device. For example, software devices do not support assigning a texture to more than one texture stage at a time. Applications should always query for device capabilities to determine which features are supported.

[\[C++, Visual Basic\]](#)

HAL Device

If the computer on which your application is running is equipped with a display adapter that supports 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 access the hardware through the hardware abstraction layer (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 **IDirect3D7::CreateDevice** method, and pass the IID_IDirect3DHALDevice value as the first parameter. For details, see [Creating a Direct3D Device](#).

[\[Visual Basic\]](#)

To create a HAL device from Visual Basic, call the **Direct3D7.CreateDevice** method, and pass the "IID_IDirect3DHALDevice" string constant as the first parameter. For details, see [Creating a Direct3D Device](#).

Note

Unlike the software-emulation RGB device, hardware devices cannot render to 8-bit render-target surfaces.

TnLHAL Device

Some new 3-D hardware is capable of performing hardware-accelerated for transformation and lighting operations, freeing the CPU of most of its workload. Where available, your application can use devices like these for optimal performance. Direct3D TnLHAL devices implement all of the transformation, lighting, and rasterizing modules in hardware.

Applications do not access 3-D cards directly. They call Direct3D functions and methods. Direct3D access the hardware through the hardware abstraction layer (HAL). If the computer that your application is running on supports the TnLHAL, it will gain the best performance by using a TnLHAL device.

[C++]

Hardware devices capable of hardware-accelerated transformation and lighting expose the D3DDEVCAPS_HWTRANSFORMANDLIGHT device capability. To create a TnLHAL device from C++, call the **IDirect3D7::CreateDevice** method, and pass the IID_IDirect3DTnLHALDevice value as the first parameter. For details, see [Creating a Direct3D Device](#).

[Visual Basic]

Hardware devices capable of hardware-accelerated transformation and lighting expose the D3DDEVCAPS_HWTRANSFORMANDLIGHT device capability. To create a TnLHAL device from Visual Basic, call the **Direct3D7.CreateDevice** method, and pass the "IID_IDirect3DTnLHALDevice" string constant as the first parameter. For details, see [Creating a Direct3D Device](#).

RGB Device

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

[C++]

Applications written in C++ create an RGB device with the **IDirect3D7::CreateDevice** method. Pass the value IID_IDirect3DRGBDevice as the first parameter. For more information, see [Creating a Direct3D Device](#).

[Visual Basic]

Applications written in Visual Basic create an RGB device with the **Direct3D7.CreateDevice** method. Pass the "IID_IDirect3DRGBDevice" string constant as the first parameter. For more information, see Creating a Direct3D Device.

Reference Rasterizer

Direct3D supports an additional device type called the reference rasterizer. Unlike an RGB 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.

The reference rasterizer is not normally enumerated by Direct3D; set the EnumReference named value in the **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Direct3D\Drivers** registry key to a nonzero **DWORD** value to enable its enumeration.

[C++]

Applications written in C++ create a reference device with the **IDirect3D7::CreateDevice** method. Pass the value IID_IDirect3DRefDevice as the first parameter. For more information, see Creating a Direct3D Device.

[Visual Basic]

Visual Basic applications create a reference device with the **Direct3D7.CreateDevice** method. Pass the "IID_IDirect3DRefDevice" string constant as the first parameter. For more information, see Creating a Direct3D Device.

Legacy Device Types

[Visual Basic]

Note

The information in this topic applies only to applications written in C++. DirectX for Visual Basic does not support these device types.

[C++]

Previous releases of DirectX supported two additional types of device, called the MMX device and the Ramp device. These devices are superseded in the current version of DirectX, but can still be accessed through legacy interfaces.

MMX Device

MMX is a special instruction set that some microprocessors support. It provides increased performance for multimedia and 3-D processing. Direct3D uses MMX support, if it is installed, to increase rendering speed.

MMX devices are not hardware-accelerated devices like HAL devices. The transformation, lighting, and rasterizing modules are completely implemented in software. However, MMX devices provide much better performance than other types of software-emulated Direct3D devices.

If your application uses the **IDirect3D2** interface, it must explicitly create an MMX device. However, beginning with the **IDirect3D3** interface, MMX and RGB devices implement exactly the same feature set (see RGB Device). If an application attempts to create an RGB device and the microprocessor supports MMX technology, Direct3D automatically creates an MMX device.

Your application creates an MMX device by invoking the **IDirect3D3::CreateDevice** method and passing the value **IID_IDirect3DMMXDevice** in the first parameter. For more information, see Creating a Direct3D Device. If the application explicitly requests that an MMX device is created and the user's computer does not support MMX technology, **CreateDevice** fails.

Ramp Device

You cannot create a Direct3D ramp device by using the **IDirect3D3** or **IDirect3D7** interfaces, nor can you query an existing ramp device for the **IDirect3DDevice3** or **IDirect3DDevice7** interfaces. In short, ramp devices do not support any multiple texture blending options. For emulation of these features, use the MMX or RGB software emulation devices.

A ramp device is a software-emulated device that provides monochrome lighting. Applications can select it when the user's computer does not have sufficient processing power to support any other type of Direct3D devices. Ramp mode uses the full range of the transformation module but does not support colored lighting. A ramp mode device uses the Direct3D lighting module for monochrome gray-scale lighting only. Therefore, lights in ramp mode have intensity but no color. The gray-scale value is stored in the blue component of the light color.

Colored materials and textures can be used in ramp mode. Direct3D uses the material or texture color as its base color. If white light (full-intensity light) is shining on the material or texture, the base color is used. However, if the light strength is less than full intensity, Direct3D mixes gray or black into the color of the material or texture. If your application uses textures in ramp mode, it must set the **D3DLIGHTSTATE_MATERIAL** member of the legacy **D3DLIGHTSTATETYPE** enumerated type. Only 8-bit textures can be used in ramp mode.

To control the number of color values available for a material or texture in ramp mode, your application must set the **dwRampSize** member of the legacy **D3DMATERIAL** structure when it creates its materials. Direct3D uses the material and texture color as the base color. The value in the **dwRampSize** member determines how many gradients of the base color are available, depending on the brightness of the light. Direct3D creates a color palette with the number of entries (1-based) specified in the **dwRampSize** member. Since the maximum possible number of palette entries is less than 256 (256 minus the reserved colors that Windows uses), your application should specify the minimum number of gray-scale values required for the application.

For best results, make the ramp size for most or all of your application's materials the same value. When Direct3D runs out of palette entries, it searches through the existing materials to find the closest color match. Only materials with the same ramp size can be considered a match.

Ramp devices are provided primarily for backward compatibility with legacy applications. In general, computers that do not have sufficient processing power to support RGB devices are not suited to 3-D applications.

An application creates a ramp device by invoking the **IDirect3D2::CreateDevice** method and passing the value **IID_IDirect3DRampDevice** as the first parameter. For more information, see [Creating a Direct3D Device](#).

Device Interfaces

[\[Visual Basic\]](#)

Note

The information in this topic applies only to applications written in C++. DirectX for Visual Basic does not use COM interfaces, nor does it support rendering with execute-buffers.

[\[C++\]](#)

Applications written in C++ use a device interface to manipulate a **Direct3DDevice** object's rendering states, lighting states, and to perform rendering operations. There are four COM interfaces for devices in Direct3D: **IDirect3DDevice**, **IDirect3DDevice2**, **IDirect3DDevice3**, and **IDirect3DDevice7**. Underlying these interfaces is one of two possible COM objects that expose either all of the first three interfaces or only the **IDirect3DDevice7** interface. The device object—and thus the interfaces—that your application uses depends entirely on the method by which the device is created. For more information, see [Creating a Direct3D Device](#). Devices that expose the legacy **IDirect3DDevice**, **IDirect3DDevice2**, and **IDirect3DDevice3** interfaces represent features and rendering capabilities exposed in versions of DirectX earlier than DirectX 7.0. This documentation refers to these objects as old devices.

Objects that expose the **IDirect3DDevice7** interface, which this documentation refers to as new devices, represent the latest set of rendering features. Applications cannot migrate across this interface boundary by using the **IUnknown::QueryInterface** method because the old device object is unaware of the newest features and the new device object is designed for optimal performance, rather than backward compatibility. Direct3D requires that you choose which interface your application will use ahead of time.

This documentation largely assumes that you will use the latest interface. However, some knowledge of previous implementations might be useful. The **IDirect3DDevice** interface exposed in previous versions of DirectX provides methods used for programming with execute buffers. However, execute buffers were provided primarily for backward compatibility. Subsequent versions of DirectX abandoned execute buffers in favor of the much simpler DrawPrimitive model, greatly simplifying the code required to render a scene. The DrawPrimitive rendering approach is employed by all interfaces later than the **IDirect3DDevice** interface. Direct3D Immediate Mode in DirectX 7.0 introduces the **IDirect3DDevice7** interface, which provides greater simplicity and hardware acceleration. The device object that exposes **IDirect3DDevice7** is capable of performing hardware-accelerated transformation and lighting operations when running on certain hardware. For more information, see Immediate Mode Changes for DirectX 7.0. These interfaces share some common methods, especially the interfaces that perform DrawPrimitive rendering. For more information, see Rendering.

Prior to the introduction of the **IDirect3DDevice2** interface in DirectX 5.0, Direct3D devices were actually interfaces to DirectDrawSurface objects. The **IDirect3DDevice2** and **IDirect3DDevice3** interfaces implement a device-object model in which a Direct3DDevice object is entirely separate from DirectDraw surfaces, and concepts such as lights, materials, and viewports are objects that can be associated with one or more devices to render a scene. On the other hand, the **IDirect3DDevice7** interface simplifies the Direct3D object-model further by moving lighting, material, and viewport parameters from separate objects, into the device itself.

Note

There are no interface version numbers between **IDirect3DDevice3** and **IDirect3DDevice7**.

Because devices are separated from DirectDraw surfaces and have independent lifetimes, Direct3D device objects can use various DirectDraw surfaces as render targets at different times if the application requires it. For more information, see **IDirect3DDevice7::SetRenderTarget**.

Device States

A Direct3D device is a state machine; 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 of the render states used in Direct3D Immediate Mode, 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
- State Blocks

[\[C++\]](#)

C++ applications set most device states through the **IDirect3DDevice7::SetRenderState** method.

[\[Visual Basic\]](#)

Applications developed in Visual Basic set most device states through the **Direct3DDevice7.SetRenderState** method.

Note

Device states also include user-defined clip planes, transformations, and lighting states. For more information, see [User-defined Clip Planes](#), the [Geometry Pipeline](#), and [Lighting and Materials](#).

Render States

This section introduces the concept of render states, contrasts them to texture stage states, and discusses the various render states in detail. Information in this section is organized into the following topics:

- About Render States
- Current Texture
- Antialiasing States
- Texture Addressing State
- Texture Wrapping State
- Texture Borders
- Texture Perspective State
- Texture Filtering State
- Outline and Fill States
- Shading State

- Fog States
- Alpha-blending States
- Alpha-testing States
- Texture Blending State
- Culling State
- Depth Buffering State
- Ramp State
- Subpixel Correction State
- Plane Masking State
- Color Keying State
- Render Command Batching State
- Stencil Buffer State
- Primitive Clipping State
- Lighting State
- Extent Update State
- Ambient Lighting State
- Per-vertex Color States

About Render States

Device render states control the behavior of the Direct3D device's rasterization module. By altering the attributes of the rendering state, what type of shading is used, fog attributes, and many other rasterizer operations.

[C++]

Applications written in C++ control the other characteristics of the rendering state by invoking the **IDirect3DDevice7::SetRenderState** method. The

D3DRENDERSTATETYPE enumerated type specifies all of the possible rendering states. Your application passes a value from the **D3DRENDERSTATETYPE** enumeration as the first parameter to the **IDirect3DDevice7::SetRenderState** method.

Render states also can control the style of texturing and how texture filtering is done. For DirectX 6.0 and later, all texture-related render states are superseded by corresponding features offered by the **IDirect3DDevice7::SetTextureStageState** method. Attempts to use the superseded render states with the **IDirect3DDevice7** interface will fail.

D3DRENDERSTATE_TEXTUREADDRESS,
D3DRENDERSTATE_TEXTUREADDRESSU and
D3DRENDERSTATE_TEXTUREADDRESSV

Superseded by the D3DTSS_ADDRESS, D3DTSS_ADDRESSU, D3DTSS_ADDRESSV texture stage states.

D3DRENDERSTATE_BORDERCOLOR

Superseded by the D3DTSS_BORDERCOLOR texture stage state.

D3DRENDERSTATE_TEXTUREMAG

Superseded by the D3DTSS_MAGFILTER texture stage state.

D3DRENDERSTATE_TEXTUREMIN

Superseded by the D3DTSS_MINFILTER texture stage state. Mipmap minification filtering is superseded by D3DTSS_MIPFILTER.

D3DRENDERSTATE_TEXTUREMAPBLEND

Superseded by the D3DTSS_COLOROP and D3DTSS_ALPHAOP texture stage states.

[Visual Basic]

Visual Basic applications control the characteristics of the rendering state by invoking the **Direct3DDevice7.SetRenderState** method. The

CONST_D3DRENDERSTATETYPE enumerated type specifies all of the possible rendering states. Your application passes a value from the

CONST_D3DRENDERSTATETYPE enumeration as the first parameter to the **Direct3DDevice7.SetRenderState** method.

Current Texture

[Visual Basic]

Note

Information in this topic pertains only to applications written in C++. DirectX for Visual Basic uses surface objects and the **Direct3DDevice7.SetTexture** method to assign current textures to a device.

[C++]

The D3DRENDERSTATE_TEXTUREHANDLE render state is obsolete.

Applications set the current textures by calling the **IDirect3DDevice7::SetTexture** method.

Antialiasing States

Antialiasing is a method of making lines and edges appear smoother on the screen.

Direct3D supports two ways to perform antialiasing, called edge antialiasing and full-scene antialiasing. For details about these techniques, see Antialiasing in the Common Techniques and Special Effects section.

[C++]

By default, Direct3D default doesn't perform antialiasing. The D3DRENDERSTATE_ANTIALIAS render state can be set to one of the members of **D3DANTIALIASMODE** to enable full-scene antialiasing. (The default value, D3DANTIALIAS_NONE disables full-scene antialiasing.)

To enable edge-antialiasing (which requires a second rendering pass), set the D3DRENDERSTATE_EDGEANTIALIAS render state to TRUE. To disable it, set D3DRENDERSTATE_EDGEANTIALIAS to FALSE.

[Visual Basic]

By default, Direct3D default doesn't perform antialiasing. The D3DRENDERSTATE_ANTIALIAS render state can be set to one of the members of the **CONST_D3DANTIALIASMODE** enumeration to enable full-scene antialiasing. (The default value, D3DANTIALIAS_NONE disables full-scene antialiasing.)

To enable edge-antialiasing (which requires a second rendering pass), set the D3DRENDERSTATE_EDGEANTIALIAS render state to nonzero. To disable it, set D3DRENDERSTATE_EDGEANTIALIAS to zero.

Texture Addressing State

[Visual Basic]**Note**

Information in this topic pertains only to applications written in C or C++. DirectX for Visual Basic requires applications to set the texture addressing state by way of the **Direct3DDevice7.SetTextureStageState** method.

[C++]

The D3DRENDERSTATE_TEXTUREADDRESS, D3DRENDERSTATE_TEXTUREADDRESSU, and D3DRENDERSTATE_TEXTUREADDRESSV render states are obsolete, and are superseded by the D3DTSS_ADDRESS, D3DTSS_ADDRESSU, and D3DTSS_ADDRESSV texture stage states.

Note

The **IDirect3DDevice7** interface does not recognize the legacy D3DRENDERSTATE_TEXTUREADDRESS, D3DRENDERSTATE_TEXTUREADDRESSU, and D3DRENDERSTATE_TEXTUREADDRESSV render states. Attempts to use these render states will fail.

Texture Wrapping State

[C++]

Note

The u- and v-wrapping render states `D3DRENDERSTATE_WRAPU` and `D3DRENDERSTATE_WRAPV` are obsolete and have been superseded by the `D3DRENDERSTATE_WRAP0` through `D3DRENDERSTATE_WRAP7` render states.

These wrapping render states enable and disable u- and v-wrapping for various textures in the device's multitexture cascade. 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. For a conceptual overview, see [Texture Wrapping](#).

Although `D3DRENDERSTATE_WRAPU` and `D3DRENDERSTATE_WRAPV` are superseded, the **IDirect3DDevice7** interface still recognizes them. These older render states, when passed to the **IDirect3DDevice7::SetRenderState** affect u- and v-texture wrapping for the first texture stage (stage 0).

[Visual Basic]

The `D3DRENDERSTATE_WRAP0` through `D3DRENDERSTATE_WRAP7` render states control texture wrapping for various textures in the device's multitexture cascade. Set these render states to a combination of the `D3DWRAPCOORD_0`, `D3DWRAPCOORD_1`, `D3DWRAPCOORD_2`, and `D3DWRAPCOORD_3` values from the **CONST_D3D** 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 Borders

[Visual Basic]

Note

Information in this topic pertains only to applications written in C++. DirectX for Visual Basic requires applications to set texture border behavior by setting the `D3DTSS_BORDERCOLOR` texture stage state with the **Direct3DDevice7.SetTextureStageState** method.

[C++]

The D3DRENDERSTATE_BORDERCOLOR render state is obsolete, superseded by the D3DTSS_BORDERCOLOR texture stage state supported by the **IDirect3DDevice7::SetTextureStageState** method. For more information, see About the Border Color Texture Address Mode.

Note

The D3DRENDERSTATE_BORDERCOLOR render state is not recognized by the **IDirect3DDevice7** interface. Attempts to use this render state with the **IDirect3DDevice7** interface will fail.

Texture Perspective State

[\[C++\]](#)

Applications can apply perspective correction to textures to make them fit properly onto primitives that diminish in size as they get farther away from the viewer. See D3DRENDERSTATE_TEXTUREPERSPECTIVE.

The following code fragment illustrates the process of enabling texture perspective correction:

```
// This code fragment assumes that lpD3DDevice is a valid pointer to
// an IDirect3DDevice7 interface.

// Enable texture perspective.
lpD3DDevice7->SetRenderState(D3DRENDERSTATE_TEXTUREPERSPECTIVE,
                             TRUE);
```

For the **IDirect3DDevice3** and **IDirect3DDevice7** interfaces, the default value is TRUE to enable perspective correct texture mapping. For earlier interfaces, the default is FALSE. Note that many 3-D adapters apply texture perspective correction unconditionally. Perspective correction must be enabled to use w-based fog, and w-buffers. For more information, see Eye-relative vs. Z-Based Depth in the Fog section, and Enabling Depth Buffering in the Depth Buffers section.

[\[Visual Basic\]](#)

Applications can apply perspective correction to textures to make them fit properly onto primitives that diminish in size as they get farther away from the viewer. See D3DRENDERSTATE_TEXTUREPERSPECTIVE.

The following code fragment illustrates the process of enabling texture perspective correction:

```
' This code fragment assumes that d3dD3DDevice is a valid reference to
' a Direct3DDevice7 object.

' Enable texture perspective.
Call d3dDevice7.SetRenderState(D3DRENDERSTATE_TEXTUREPERSPECTIVE, _
```

True)

The default value is True to enable perspective correct texture mapping.

Note that many 3-D adapters apply texture perspective correction unconditionally. Perspective correction must be enabled to use w-based fog, and w-buffers. For more information, see Eye-relative vs. Z-Based Depth in the Fog section, and Enabling Depth Buffering in the Depth Buffers section.

Texture Filtering State

[\[Visual Basic\]](#)

Note

Information in this topic pertains only to applications written in C++. DirectX for Visual Basic requires applications to set texture-filtering behavior by using the **Direct3DDevice7.SetTextureStageState** method.

[\[C++\]](#)

Note

The legacy render states, D3DRENDERSTATE_TEXTUREMAG, D3DRENDERSTATE_TEXTUREMIN, and D3DRENDERSTATE_ANISOTROPY, are obsolete with the release of DirectX 7.0.

Applications must use the D3DTSS_MAGFILTER, D3DTSS_MINFILTER, and D3DTSS_MAXANISOTROPY texture-stage states to control texture filtering.

Outline and Fill States

[\[C++\]](#)

Primitives that have no textures are rendered with the color specified by their material, or with the colors specified for the vertices, if any. The method used to fill them can be selected with the **D3DFILLMODE** enumerated type. See D3DRENDERSTATE_FILLMODE.

If you want your application to enable dithering, it must pass the D3DRENDERSTATE_DITHERENABLE enumerated value as the first parameter to **IDirect3DDevice7::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. This can be controlled using the D3DRENDERSTATE_LASTPIXEL enumerated value. However, this setting should not be altered without some forethought. Under some conditions, suppressing the rendering of the last pixel can cause unsightly gaps between primitives.

By default, Direct3D devices use a solid outline for primitives. The outline pattern can be changed using the **D3DLINEPATTERN** structure. See **D3DRENDERSTATE_LINEPATTERN**.

Note

The legacy render states **D3DRENDERSTATE_ROP2**, **D3DRENDERSTATE_STIPPLEENABLE**, and **D3DRENDERSTATE_STIPPLEPATTERN00** through **D3DRENDERSTATE_STIPPLEPATTERN31** are obsolete with the release of DirectX 7.0.

[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. The method used to fill them can be selected with the **CONST_D3DFILLMODE** enumerated type. See **D3DRENDERSTATE_FILLMODE**.

If you want your application to enable dithering, it must pass the **D3DRENDERSTATE_DITHERENABLE** enumerated value as the first parameter to **Direct3DDevice7.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. This can be controlled using the **D3DRENDERSTATE_LASTPIXEL** enumerated value. However, this setting should not be altered without some forethought. Under some conditions, suppressing the rendering of the last pixel can cause unsightly gaps between primitives.

DirectX for Visual Basic does not support the **D3DRENDERSTATE_LINEPATTERN** render state.

Shading State

[C++]

Direct3D supports both flat and Gouraud shading. The default is Gouraud shading. To control the current shading mode, your C++ application uses the **D3DSHADEMODE** enumerated type. See **D3DRENDERSTATE_SHADEMODE**.

The following C++ code fragment demonstrates the process of setting the shading state to flat shading mode:

```
// This code fragment assumes that lpD3DDevice is a valid pointer to
// a IDirect3DDevice7 interface.

// Set the shading state.
lpD3DDevice->SetRenderState(D3DRENDERSTATE_SHADEMODE,
```

D3DSHADE_FLAT);

[Visual Basic]

Direct3D supports both flat and Gouraud shading. The default is Gouraud shading. To control the current shading mode from a Visual Basic application, use the **CONST_D3DSHADEMODE** enumerated type. See **D3DRENDERSTATE_SHADEMODE**.

The following Visual Basic code fragment demonstrates the process of setting the shading state to flat shading mode:

```
' This example assumes that D3DDevice is a valid reference
```

```
' to a Direct3DDevice7 object.
```

```
' Set the shading state.
```

```
Call D3DDevice.SetRenderState(D3DRENDERSTATE_SHADEMODE, D3DSHADE_FLAT)
```

Fog States

Fog effects can give a 3-D scene greater realism. Fog effects can be used for more than simulating fog. They can also be used to decrease the clarity of a scene with distance. This mirrors what happens in the real world. As objects get more distant from the viewer, their detail becomes 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 to the formula.

You enable fog by setting the **D3DRENDERSTATE_FOGENABLE** render state to **TRUE**. The fog color can be set to any **D3DCOLOR** value (the alpha component of the fog color is ignored). See **D3DRENDERSTATE_FOGCOLOR**.

The **D3DRENDERSTATE_FOGTABLEMODE** and **D3DRENDERSTATE_FOGVERTEXMODE** render states control the fog formula applied for fog calculations, and indirectly control which type of fog is applied at all. Both render states can be set to one of the members 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 will apply only pixel fog effects.

The **D3DRENDERSTATE_FOGSTART** and **D3DRENDERSTATE_FOGEND** render states control fog formula parameters for the **D3DFOG_LINEAR** mode. The **D3DRENDERSTATE_FOGDENSITY** render state controls fog density in the exponential fog modes.

For more information, see Fog Parameters.

[Visual Basic]

In Visual Basic, applications control fog through device rendering states. The **CONST_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 to that formula.

You enable fog by setting the **D3DRENDERSTATE_FOGENABLE** render state to **True**. The fog color can be set to any value of type **Long** returned by the **DirectX7.CreateColorRGB** method. See **D3DRENDERSTATE_FOGCOLOR**.

The **D3DRENDERSTATE_FOGTABLEMODE** and **D3DRENDERSTATE_FOGVERTEXMODE** render states control the fog formula that the system applies for fog calculations, and indirectly control which type of fog is applied at all. Both render states can be set to one of the members 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 will apply only pixel fog effects.

The **D3DRENDERSTATE_FOGSTART** and **D3DRENDERSTATE_FOGEND** render states control fog formula parameters for the **D3DFOG_LINEAR** mode. The **D3DRENDERSTATE_FOGDENSITY** render state controls fog density in the exponential fog modes.

For more information, see Fog Parameters.

Alpha-Blending States

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 Multipass Texture Blending.

[C++]

Applications written in C++ use the **D3DRENDERSTATE_ALPHABLENDENABLE** render state to enable alpha transparency blending. The Direct3D API allows many types of alpha blending. However, it is important to note the user's 3-D hardware may not support all of the blending states allowed by Direct3D.

The type of alpha blending that is done depends on the **D3DRENDERSTATE_SRCBLEND** and **D3DRENDERSTATE_DESTBLEND** render states. Source and destination blend states are used in pairs. The following code fragment demonstrates how the source blend state is set to **D3DBLEND_SRCCOLOR** and the destination blend state is set to **D3DBLEND_INVSRCOLOR**.

```
// This code fragment assumes that lpD3DDevice is a valid pointer to
```

```
// an IDirect3DDevice7 interface.

// Set the source blend state.
lpD3DDevice->SetRenderState(D3DRENDERSTATE_SRCBLEND,
                           D3DBLEND_SRCCOLOR);

// Set the destination blend state.
lpD3DDevice->SetRenderState(D3DRENDERSTATE_DESTBLEND,
                           D3DBLEND_INVSRCOLOR);
```

As a result of the calls in the preceding code fragment, Direct3D performs a linear blend between the source color (the color of the primitive being 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 it 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 controlling 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.

Color light mapping can be achieved by setting the source alpha blending state to D3DBLEND_ZERO and the destination blend state to D3DBLEND_SRCCOLOR.

Direct3D devices provide alpha value stippling if it is supported by the display hardware. See D3DRENDERSTATE_STIPPLEDALPHA. If your application creates an RGB or ramp software emulation device, Direct3D ignores this enumerated value.

[\[Visual Basic\]](#)

Visual Basic applications use the D3DRENDERSTATE_ALPHABLENDENABLE render state to enable alpha transparency blending. The Direct3D API allows many types of alpha blending. However, it is important to note the user's 3-D hardware may not support all of the blending states allowed by Direct3D.

The type of alpha blending that is done depends on the D3DRENDERSTATE_SRCBLEND and D3DRENDERSTATE_DESTBLEND render states. Source and destination blend states are used in pairs. The following code fragment demonstrates how the source blend state is set to D3DBLEND_SRCCOLOR and the destination blend state is set to D3DBLEND_INVSRCOLOR.

' This code fragment assumes that D3DDevice contains a

' reference to a Direct3DDevice7 object.

' Set the source blend state.

```
Call D3DDevice.SetRenderState(D3DRENDERSTATE_SRCBLEND, _  
                             D3DBLEND_SRCCOLOR);
```

' Set the destination blend state.

```
Call D3DDevice.SetRenderState(D3DRENDERSTATE_DESTBLEND, _  
                             D3DBLEND_INVSRCCOLOR);
```

As a result of the calls in the preceding code fragment, Direct3D performs a linear blend between the source color (the color of the primitive being 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 it 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 controlling 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.

Color light mapping can be achieved by setting the source alpha blending state to D3DBLEND_ZERO and the destination blend state to D3DBLEND_SRCCOLOR.

Direct3D devices provide alpha value stippling if it is supported by the display hardware. See D3DRENDERSTATE_STIPPLEDALPHA. If your application creates an RGB or ramp software emulation device, Direct3D ignores this enumerated value.

Alpha-Testing States

[C++]

C++ applications can use alpha testing to control when pixels are written to the render-target surface. By using the D3DRENDERSTATE_ALPHATESTENABLE enumerated value, your application sets the current 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 doesn't, Direct3D ignores it. Select the alpha test function with the D3DRENDERSTATE_ALPHAFUNC enumerated value. Your application can set a reference alpha value for all pixels to be compared against by using the D3DRENDERSTATE_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 already 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 example checks to see 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 pd3dDeviceDesc is a
// D3DDEVICEDESC7 structure that was filled with a
// previous call to IDirect3DDevice7::GetCaps.
if (pd3dDeviceDesc.dpcTriCaps.dwAlphaCmpCaps & D3DPCMPCAPS_GREATEREQUAL)
{
    dev->SetRenderState( D3DRENDERSTATE_ALPHAREF, (DWORD)0x00000001);
    dev->SetRenderState( D3DRENDERSTATE_ALPHATESTENABLE, TRUE );
    dev->SetRenderState( D3DRENDERSTATE_ALPHAFUNC, D3DCMP_GREATEREQUAL );
}
```

// If the comparison isn't 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 **IDirect3DDevice7::GetCaps** method. After retrieving the device capabilities, check the **dwAlphaCmpCaps** member of the **D3DPRIMCAPS** structure (contained by the associated **D3DDEVICEDESC7** structure) for the desired comparison function. If the **dwAlphaCmpCaps** member contains only the **D3DPCMPCAPS_ALWAYS** capability or only the **D3DPCMPCAPS_NEVER** capability, the driver does not support alpha tests at all.

[\[Visual Basic\]](#)

Applications written in Visual Basic use alpha testing to control when pixels are written to the render-target surface. By using the **D3DRENDERSTATE_ALPHATESTENABLE** enumerated value, your application sets the current 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 doesn't, Direct3D ignores it. Select the alpha test function with the **D3DRENDERSTATE_ALPHAFUNC** render state. Your application can set a reference alpha value for all pixels to be compared against by using the **D3DRENDERSTATE_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 already 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 example checks to see 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 d3dDeviceDesc is a
' D3DDEVICEDESC7 type that was filled with a
' previous call to Direct3DDevice7.GetCaps.
If (d3dDeviceDesc.dpcTriCaps.lAlphaCmpCaps And D3DPCMPCAPS_GREATEREQUAL)
Then
    Call dev.SetRenderState(D3DRENDERSTATE_ALPHAREF, &H1)
    Call dev.SetRenderState(D3DRENDERSTATE_ALPHATESTENABLE, True)
    Call dev.SetRenderState(D3DRENDERSTATE_ALPHAFUNC,
D3DCMP_GREATEREQUAL)
End If

' If the comparison isn't 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 **Direct3DDevice7.GetCaps** method. After retrieving the device capabilities, check the **lAlphaCmpCaps** member of the **D3DPRIMCAPS** type (contained by the associated **D3DDEVICEDESC7** type) for the desired comparison function. If the **lAlphaCmpCaps** member contains only the **D3DPCMPCAPS_ALWAYS** capability or only the **D3DPCMPCAPS_NEVER** capability, the driver does not support alpha tests at all.

Texture Blending State

[Visual Basic]

Note

Information in this topic pertains only to applications written in C++. DirectX for Visual Basic requires applications to control texture blending by using the **Direct3DDevice7.SetTextureStageState** method with the **D3DTSS_COLOROP** and **D3DTSS_ALPHAOP** texture stage states.

[C++]

The **D3DRENDERSTATE_TEXTUREMAPBLEND** render state is superseded by the texture blending states available through the **D3DTSS_COLOROP** and **D3DTSS_ALPHAOP** texture stage states. For more information, see [Multiple Texture Blending](#).

Note

The **D3DRENDERSTATE_TEXTUREMAPBLEND** render state is obsolete, and is not recognized by the **IDirect3DDevice7** interface. Attempts to use this render state with the **IDirect3DDevice7** interface will fail.

Culling State

As Direct3D renders primitives, it culls those primitives that are facing away from the viewer.

[C++]

You can set the culling mode from a C++ application by using the **D3DCULL** enumerated type. See D3DRENDERSTATE_CULLMODE. 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 fragment assumes that lpD3DDevice is a valid pointer to
// an IDirect3DDevice interface.

// Set the culling state.
lpD3DDevice->SetRenderState(D3DRENDERSTATE_CULLMODE,
                           D3DCULL_CW);
```

[Visual Basic]

Visual Basic applications set the culling mode by using D3DRENDERSTATE_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 fragment assumes that D3DDevice contains a valid
' reference to a Direct3DDevice 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. For a conceptual overview, see What Are Depth Buffers?. By default, Direct3D does not use depth buffering.

[C++]

C++ applications update the depth buffering state with the D3DRENDERSTATE_ZENABLE render state, using one of the members of the **D3DZBUFFERTYPE** enumeration to specify the new state value.

If, for some reason, your application needs to prevent Direct3D from writing to the depth buffer, it can use the `D3DRENDERSTATE_ZWRITEENABLE` enumerated value, specifying `FALSE` as the second parameter for the call to **IDirect3DDevice7::SetRenderState**.

The following code fragment shows how the depth buffer state is set to enable z-buffering:

```
// This code fragment assumes that lpD3DDevice is a valid pointer to
// a and IDirect3DDevice7 interface.

// Enable z-buffering.
lpD3DDevice->SetRenderState(D3DRENDERSTATE_ZENABLE,
                           D3DZB_TRUE); // D3DZB_TRUE is the same as TRUE
```

Your application can also use the `D3DRENDERSTATE_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 `D3DRENDERSTATE_ZBIAS`).

[\[Visual Basic\]](#)

You can update the depth buffering state from a Visual Basic application with the `D3DRENDERSTATE_ZENABLE` render state, using one of the members of the **CONST_D3DZBUFFERTYPE** enumeration to specify the new state value.

If, for some reason, your application needs to prevent Direct3D from writing to the depth buffer, it can use the `D3DRENDERSTATE_ZWRITEENABLE` render state, specifying `D3DZB_FALSE` as the second parameter for the call to **Direct3DDevice7.SetRenderState**.

The following Visual Basic code fragment shows how the depth buffer state is set to enable z-buffering:

```
' This code fragment assumes that D3DDevice contains a valid
' reference to a Direct3DDevice7 object.

' Enable z-buffering.
Call D3DDevice.SetRenderState(D3DRENDERSTATE_ZENABLE, _
                             D3DZB_TRUE)
```

Your application can also use the `D3DRENDERSTATE_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 D3DRENDERSTATE_ZBIAS).

Ramp State

The D3DRENDERSTATE_MONOENABLE render state is not currently used. Ramp lighting is automatically enabled when an application uses a ramp device. For more information, see Legacy Device Types.

Subpixel Correction State

The D3DRENDERSTATE_SUBPIXEL and D3DRENDERSTATE_SUBPIXELX render states are not used, and are ignored by the system.

Plane Masking State

[\[Visual Basic\]](#)

Note

Information in this topic pertains only to applications written in C++.

[\[C++\]](#)

Note

The D3DRENDERSTATE_PLANEMASK render state is obsolete with the release of DirectX 7.0. To disable writes to the color buffer by using alpha blending, set D3DRENDERSTATE_SRCBLEND to D3DBLEND_ZERO and D3DRENDERSTATE_DESTBLEND to D3DBLEND_ONE.

Color Keying State

Setting a color key instructs Direct3D to treat the key color as transparent. When Direct3D applies a texture that was created with the DDS_DCKSRCBLT flag to a primitive, all texels that match the key color are not rendered on the primitive. Note that any textures that were not created with the DDS_DCKSRCBLT flag will not display color-key effects, even if they contain the color key.

[\[C++\]](#)

C++ applications set color keys with the **IDirectDrawSurface7::SetColorKey** method for the surface that will be using the color key (in this case, the render-target surface). However, color keying can be toggled on and off by calling the **IDirect3DDevice7::SetRenderState** method. Set the first parameter to D3DRENDERSTATE_COLORKEYENABLE. Your application should set the

second parameter to TRUE to enable color keying and FALSE to disable it. The default is FALSE.

```
// This code fragment assumes that lpD3DDevice is a valid pointer to
// an IDirect3DDevice7 interface.

// Disable color keying.
lpD3DDevice->SetRenderState(D3DRENDERSTATE_COLORKEYENABLE,
                           FALSE);
```

[\[Visual Basic\]](#)

Applications written in Visual Basic set color keys with the **DirectDrawSurface7.SetColorKey** method for the surface that will be using the color key (in this case, the render-target surface). However, color keying can be toggled on and off by calling the **Direct3DDevice7.SetRenderState** method. Set the first parameter to D3DRENDERSTATE_COLORKEYENABLE. Your application should set the second parameter to True to enable color keying and False to disable it. The default is False.

```
' This code fragment assumes that D3DDevice contains a valid reference
' to a Direct3DDevice7 object.

' Disable color keying.
Call D3DDevice.SetRenderState(D3DRENDERSTATE_COLORKEYENABLE, _
                             FALSE)
```

Render Command Batching State

[\[Visual Basic\]](#)

Information in this topic pertains only to applications in C++.

[\[C++\]](#)

Note

This render state obsolete, and is only useful to legacy applications that render with texture handles (using the **IDirect3DDevice2** interface). Batched primitives are implicitly flushed when rendering with the **IDirect3DDevice7** interface.

Stencil Buffer State

Applications use the stencil buffer to determine whether or not 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

IDirect3DDevice7::SetRenderState method. Pass

D3DRENDERSTATE_STENCILENABLE as the value of the first parameter. Set the value of the second parameter to TRUE or FALSE to enable or disable it respectively.

Set the comparison function that Direct3D uses to perform the stencil test by invoking the **IDirect3DDevice7::SetRenderState** method. Set the value of the first parameter to D3DRENDERSTATE_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 it by calling the **IDirect3DDevice7::SetRenderState** method. Pass D3DRENDERSTATE_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 any pixel, it does a bitwise **AND** of the stencil reference value and a stencil mask value. The result is then compared against the contents of the stencil buffer using the stencil comparison function. Your application can set the stencil mask. Use the **IDirect3DDevice7::SetRenderState** method, and pass D3DRENDERSTATE_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, invoke the

IDirect3DDevice7::SetRenderState method and pass

D3DRENDERSTATE_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 the **IDirect3DDevice7::SetRenderState** method and pass D3DRENDERSTATE_STENCILZFAIL as the first parameter and use a member of the **D3DSTENCILOP** enumerated type for the second parameter.

In addition, your program can control what Direct3D does when both the stencil test and the z-buffer test pass. Use the **IDirect3DDevice7::SetRenderState** method and pass D3DRENDERSTATE_STENCILPASS as the first parameter. Again, the second parameter must be a member of the **D3DSTENCILOP** enumerated type.

[\[Visual Basic\]](#)

You enable or disable stenciling in Visual Basic by calling the

Direct3DDevice7.SetRenderState method. Pass

D3DRENDERSTATE_STENCILENABLE as the value of the first parameter. Set the value of the second parameter to True or False to enable or disable it respectively.

Set the comparison function that Direct3D uses to perform the stencil test by invoking the **Direct3DDevice7.SetRenderState** method. Set the value of the first parameter to D3DRENDERSTATE_STENCILFUNC. Pass a member of the **CONST_D3DCMPFUNC** enumeration 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 it by calling the **Direct3DDevice7.SetRenderState** method. Pass **D3DRENDERSTATE_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 any pixel, it performs a bitwise **And** operation of the stencil reference value and a stencil mask value. The result is then compared against the contents of the stencil buffer using the stencil comparison function. Your application can set the stencil mask by using the **D3DRENDERSTATE_STENCILMASK** render state. Set the render state value to the new stencil mask.

To set the action that Direct3D takes when the stencil test fails, invoke the **Direct3DDevice7.SetRenderState** method and pass **D3DRENDERSTATE_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 the **Direct3DDevice7.SetRenderState** method and pass **D3DRENDERSTATE_STENCILZFAIL** as the first parameter and use a member of the **D3DSTENCILOP** enumeration for the second parameter.

In addition, your program can control what Direct3D does when both the stencil test and the z-buffer test pass. Use the **Direct3DDevice7.SetRenderState** method and pass **D3DRENDERSTATE_STENCILPASS** as the first parameter. Again, the second parameter must be a member of the **D3DSTENCILOP** enumeration.

Primitive Clipping State

[C++]

Direct3D can clip primitives that render partially outside the viewport. When using C++, Direct3D clipping is controlled by the **D3DRENDERSTATE_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 **IDirect3DVertexBuffer7::ProcessVertices** and **IDirect3DVertexBuffer7::ProcessVerticesStrided** methods accept their own flags to control primitive clipping during vertex processing.

Note

This render state is analogous, albeit reciprocal, to the **D3DDP_DONOTCLIP** flag used in legacy rendering methods. This render state supersedes the **D3DDP_DONOTCLIP** flag for DirectX 7.0. Where the **D3DDP_DONOTCLIP** flag disabled clipping on a per-call basis, this render state affects all rendering calls until the state value is again changed by the application.

[Visual Basic]

Direct3D can clip primitives that render partially outside the viewport. In a Visual Basic application, Direct3D clipping is controlled by the `D3DRENDERSTATE_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

`IDirect3DVertexBuffer7::ProcessVertices` method accepts flags to control primitive clipping during vertex processing.

Lighting State

Applications that use the Direct3D geometry pipeline can enable or disable lighting calculations. Only vertices that contain a vertex normal will be properly lit; vertices with no normal will use a dot product of 0 in all lighting computations. This means that a vertex that doesn't use a normal will effectively receive no light. For more information, see *Mathematics of Direct3D Lighting*.

[C++]

Applications enable Direct3D lighting by setting the `D3DRENDERSTATE_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

`IDirect3DVertexBuffer7::ProcessVertices` and

`IDirect3DVertexBuffer7::ProcessVerticesStrided` methods accept their own flags to control lighting calculations during vertex processing.

Note

This render state is analogous, albeit reciprocal, to the `D3DDP_DONOTLIGHT` flag used in legacy rendering methods. This render state supersedes the `D3DDP_DONOTLIGHT` flag for DirectX 7.0. Where the `D3DDP_DONOTLIGHT` flag disabled lighting on a per-call basis, this render state affects all rendering calls until the state value is again changed by the application.

[Visual Basic]

Applications enable Direct3D lighting by setting the `D3DRENDERSTATE_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 **IDirect3DVertexBuffer7::ProcessVertices** and **IDirect3DVertexBuffer7::ProcessVerticesStrided** methods accept their own flags to control lighting calculations during vertex processing.

Note

This render state is analogous, albeit reciprocal, to the D3DDP_DONOTLIGHT flag used in legacy rendering methods. This render state supersedes the D3DDP_DONOTLIGHT flag for DirectX 7.0. Where the D3DDP_DONOTLIGHT flag disabled lighting on a per-call basis, this render state affects all rendering calls until the state value is again changed by the application.

Extent Update State

[C++]

Direct3D supports updating the viewport extents to reflect the area of the viewport affected by rendering methods. In an application written with C++, the D3DRENDERSTATE_EXTENTS render state controls extents updates made by the system. The default setting for this state, FALSE, disables extent updates. Applications that intend to use the **IDirect3DDevice7::GetClipStatus** and **IDirect3DDevice7::SetClipStatus** methods to manipulate the viewport extents can enable extents updates by setting D3DRENDERSTATE_EXTENTS to TRUE.

The extents updates render state is entirely independent of extents updates that can be performed with a vertex buffer. The **IDirect3DVertexBuffer7::ProcessVertices** and **IDirect3DVertexBuffer7::ProcessVerticesStrided** methods accept their own flags to control extents updates during vertex processing.

Note

This render state is analogous, albeit reciprocal, to the D3DDP_DONOTUPDATEEXTENTS flag used in legacy rendering methods. This render state supersedes the D3DDP_DONOTUPDATEEXTENTS flag for DirectX 7.0. Where the D3DDP_DONOTUPDATEEXTENTS flag disabled extent updates on a per-call basis, this render state affects all rendering calls until the state value is again changed by the application.

[Visual Basic]

Direct3D supports updating the viewport extents to reflect the area of the viewport affected by rendering methods. A Visual Basic application can use the D3DRENDERSTATE_EXTENTS render state to control extents updates made by the system. The default setting for this state, False, disables extent updates. Applications that intend to use the **Direct3DDevice7.GetClipStatus** and

Direct3DDevice7.SetClipStatus methods to manipulate the viewport extents can enable extents updates by setting D3DRENDERSTATE_EXTENTS to True.

The extents updates render state is entirely independent of extents updates that can be performed with a vertex buffer. The **Direct3DVertexBuffer7.ProcessVertices** method accepts its own flags to control extents updates during vertex processing.

Ambient Lighting State

Ambient light is the surrounding light that comes from all directions. For specific information on how Direct3D used ambient light, see Direct Light vs. Ambient Light, and Mathematics of Direct3D Lighting.

[C++]

Your C++ application sets the color of the ambient lighting by invoking the **IDirect3DDevice7::SetRenderState** method and passing the enumerated value D3DRENDERSTATE_AMBIENT as the first parameter. The second parameter is a color in RGBA format. The default is zero.

```
// This code fragment assumes that lpD3DDevice is a valid pointer to
// an IDirect3DDevice7 interface.

// Set the ambient light.
D3DCOLOR d3dcAmbientLightColor = D3DRGBA(1.0f,1.0f,1.0f,1.0f);
lpD3DDevice->SetRenderState(D3DRENDERSTATE_AMBIENT,
                           d3dcAmbientLightColor);
```

[Visual Basic]

A Visual Basic application sets the color of the ambient lighting by invoking the **Direct3DDevice7.SetRenderState** method and passing the enumerated value D3DRENDERSTATE_AMBIENT as the first parameter. The second parameter is a color in RGBA format. The default is zero.

```
' This code fragment assumes that D3DDevice contains a valid
' reference to a Direct3DDevice7 object, and dx contains a
' valid reference to a DirectX7 object.
Dim AmbientLightColor As Long

' Set the ambient light.
AmbientLightColor = dx.CreateColorRGBA(1#, 1#, 1#, 1#)
Call D3DDevice.SetRenderState(D3DRENDERSTATE_AMBIENT, _
                             AmbientLightColor)
```

Per-vertex Color States

[C++]

Vertices of type **D3DVERTEX**, **D3DLVERTEX**, and **D3DTLVERTEX** cannot contain both color and normal information. Because the default shading mode is Gouraud shading, which depends on both vertex color and normal information, Direct3D did not use vertex color in the lighting calculations.

With the introduction of the flexible vertex format in DirectX 5.0, vertices can contain both vertex color and vertex normal information. By default, Direct3D uses this information when it calculates lighting. If you want your application to disable the use of vertex color lighting information, invoke the

IDirect3DDevice7::SetRenderState method and pass

D3DRENDERSTATE_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

D3DRENDERSTATE_DIFFUSEMATERIALSOURCE,

D3DRENDERSTATE_SPECULARMATERIALSOURCE,

D3DRENDERSTATE_AMBIENTMATERIALSOURCE, and

D3DRENDERSTATE_EMISSIVEMATERIALSOURCE render states control color sources for the diffuse, specular, ambient, and emissive color component sources.

Each of these states 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]

Vertices of type **D3DVERTEX**, **D3DLVERTEX**, and **D3DTLVERTEX** cannot contain both color and normal information. Because the default shading mode is Gouraud shading, which depends on both vertex color and normal information, Direct3D did not use vertex color in the lighting calculations.

When using flexible vertex formats, as opposed to the aforementioned vertex types, vertices can contain both vertex color and vertex normal information. By default, Direct3D uses this information when it calculates lighting. If you want your application to disable the use of vertex color lighting information, invoke the

Direct3DDevice7.SetRenderState method and pass

D3DRENDERSTATE_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

D3DRENDERSTATE_DIFFUSEMATERIALSOURCE,

D3DRENDERSTATE_SPECULARMATERIALSOURCE,

D3DRENDERSTATE_AMBIENTMATERIALSOURCE, and

D3DRENDERSTATE_EMISSIVEMATERIALSOURCE render states control color

sources for the diffuse, specular, ambient, and emissive color component sources. Each of these states can be set to members of the **CONST_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.

State Blocks

A state block in Direct3D Immediate Mode is a group of device states—render states, lighting and material parameters, transformation states, texture stage states, and current texture information—taken as a snapshot of the device's current state or explicitly recorded, that 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 can simply make applying device states easier.

The following topics in this section describe the key concepts of device state blocks, and provide information about how applications use them:

- State Block Handles
- Recording State Blocks
- Capturing State Blocks
- Applying State Blocks
- Creating Predefined State Blocks
- Deleting State Blocks

State Block Handles

State block handles are 32-bit values that uniquely identify a cached group of device states, and the associated state values, within the rendering device. These handles are used to apply states to the device, to capture the current state of the rendering device, and to delete the cached states when they are no longer needed.

[C++]

In C++, you receive a state-block handle when you finish recording a state block by calling the **IDirect3DDevice7::EndStateBlock** method, and when you capture a predefined set of device state data by calling the **IDirect3DDevice7::CreateStateBlock** method.

[Visual Basic]

In Visual Basic, you receive a state-block handle when you finish recording a state block by calling the **Direct3DDevice7.EndStateBlock** method, and when you capture a predefined set of device state data by calling the **Direct3DDevice7.CreateStateBlock** method.

Recording State Blocks

[C++]

The **IDirect3DDevice7** interface offers the **IDirect3DDevice7::BeginStateBlock** method to record device states into a state block, as your application calls for them. The **BeginStateBlock** method causes the system to start recording device state changes into a state block, rather than applying them to the device. After you call **BeginStateBlock**, calls to any of the following methods will be recorded into a device state block:

- **IDirect3DDevice7::LightEnable**
- **IDirect3DDevice7::SetClipPlane**
- **IDirect3DDevice7::SetLight**
- **IDirect3DDevice7::SetMaterial**
- **IDirect3DDevice7::SetRenderState**
- **IDirect3DDevice7::SetTexture**
- **IDirect3DDevice7::SetTextureStageState**
- **IDirect3DDevice7::SetTransform**
- **IDirect3DDevice7::SetViewport**

When you're done recording the state block, notify the system to stop recording by calling the **IDirect3DDevice7::EndStateBlock** method. The **EndStateBlock** method places the handle of the state block you recorded into the variable whose address you pass in the *lpdwBlockHandle* parameter. Your application will use 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 **Direct3DDevice7** object offers the **Direct3DDevice7.BeginStateBlock** method to record device states into a state block, as your application calls for them. The **BeginStateBlock** method causes the system to start recording device state changes into a state block, rather than applying them to the device. After you call **BeginStateBlock**, calls to any of the following methods will be recorded into a device state block:

- **Direct3DDevice7.LightEnable**
- **Direct3DDevice7.SetClipPlane**
- **Direct3DDevice7.SetLight**
- **Direct3DDevice7.SetMaterial**
- **Direct3DDevice7.SetRenderState**
- **Direct3DDevice7.SetTexture**
- **Direct3DDevice7.SetTextureStageState**
- **Direct3DDevice7.SetTransform**

- **Direct3DDevice7.SetViewport**

When you're done recording the state block, notify the system to stop recording by calling the **Direct3DDevice7.EndStateBlock** method. The **EndStateBlock** method places the handle of the state block you recorded into the variable you pass in the *blockHandle* parameter. Your application will use 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 an **BeginStateBlock/EndStateBlock** pair.

It is important to check the error code from the **EndStateBlock** method. If it fails, it is most likely because the display mode has changed (error code DDERR_SURFACELOST). 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 **IDirect3DDevice7::CaptureStateBlock** method updates the values within an existing state block to reflect the current state of the device. The method accepts a single parameter, *dwBlockHandle*, that identifies the state block which will receive the current state of the device if the call succeeds.

The **IDirect3DDevice7::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 into the state block. The following pseudo-code illustrates:

```
// The dwMyBlock variable contains a handle to a device-state block for
// a previously-recorded set of device states.

// Set the current state block.
d3dDevice->ApplyStateBlock(dwMyBlock);

// Change device states as needed.
.
.
.

// Capture the modified device state data back-into the existing block.
d3dDevice->CaptureStateBlock(dwMyBlock);
```

The **CaptureStateBlock** method doesn't capture the entire state of the device, it only updates the values for the states already within the state block. For example, imagine a state block that contains the following two operations, represented by psuedo-code:

```
SetRenderState(D3DRENDERSTATE_SHADEMODE, D3DSHADE_GOURAUD);  
SetTexture(0, lpd3dAnyTexture);
```

If your application changed either of the preceding two settings since the state block was originally recorded, the **IDirect3DDevice7::CaptureStateBlock** method will update the states within the block—and only those states—to their new values.

[\[Visual Basic\]](#)

The **Direct3DDevice7.CaptureStateBlock** method updates the values within an existing state block to reflect the current state of the device. The method accepts a single parameter, *blockHandle*, that identifies the state block which will receive the current state of the device if the call succeeds.

The **Direct3DDevice7.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 into the state block. The following pseudo-code illustrates:

```
' The MyBlock 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(MyBlock)  
  
' Change device states as needed.  
. . .  
  
' Capture the modified device state data back-into the existing block.  
Call d3dDevice.CaptureStateBlock(MyBlock)
```

The **CaptureStateBlock** method doesn't capture the entire state of the device, it only updates the values for the states already within the state block. For example, imagine a state block that contains the following two operations, represented by psuedo-code:

```
Call SetRenderState(D3DRENDERSTATE_SHADEMODE, D3DSHADE_GOURAUD)  
Call SetTexture(0, d3dAnyTexture)
```

If your application changed either of the preceding two settings since the state block was originally recorded, the **Direct3DDevice7.CaptureStateBlock** method will update the states within the block—and only those states—to their new values.

Applying State Blocks

[C++]

The **IDirect3DDevice7::ApplyStateBlock** method applies an existing state block to the device. The **ApplyStateBlock** method accepts a single parameter, *dwBlockHandle*, that identifies the state block to be applied.

Note

You cannot apply a state block while recording another state block (that is, between calls to **IDirect3DDevice7::BeginStateBlock** and **IDirect3DDevice7::EndStateBlock**). Attempts to do so will fail, returning D3DERR_INBEGINSTATEBLOCK.

[Visual Basic]

The **Direct3DDevice7.ApplyStateBlock** method applies an existing state block to the device. The **ApplyStateBlock** method accepts a single parameter, *blockHandle*, that identifies the state block to be applied.

Note

You cannot apply a state block while recording another state block (that is, between calls to **Direct3DDevice7.BeginStateBlock** and **Direct3DDevice7.EndStateBlock**). Attempts to do so will fail, returning D3DERR_INBEGINSTATEBLOCK.

Creating Predefined State Blocks

[C++]

The **IDirect3DDevice7::CreateStateBlock** method creates a new state block that contains the entire set of device states, or just 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 be captured into 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_VERTEX or D3DSBT_PIXEL). The following list summarizes the states that the system captures when you pass the D3DSBT_VERTEX or D3DSBT_PIXEL values:

- **Vertex-related States (D3DSBT_VERTEX)**

State (enabled or disabled) of all lights

Transformation matrices

User-defined clipping planes

The values set for the following render states:

D3DRENDERSTATE_AMBIENT
D3DRENDERSTATE_AMBIENTMATERIALSOURCE
D3DRENDERSTATE_CLIPPING
D3DRENDERSTATE_CLIPPLANEENABLE
D3DRENDERSTATE_COLORVERTEX
D3DRENDERSTATE_CULLMODE
D3DRENDERSTATE_DIFFUSEMATERIALSOURCE
D3DRENDERSTATE_EMISSIVEMATERIALSOURCE
D3DRENDERSTATE_EXTENTS
D3DRENDERSTATE_FOGCOLOR
D3DRENDERSTATE_FOGDENSITY
D3DRENDERSTATE_FOGENABLE
D3DRENDERSTATE_FOGEND
D3DRENDERSTATE_FOGSTART
D3DRENDERSTATE_FOGTABLEMODE
D3DRENDERSTATE_FOGVERTEXMODE
D3DRENDERSTATE_LIGHTING
D3DRENDERSTATE_LOCALVIEWER
D3DRENDERSTATE_NORMALIZENORMALS
D3DRENDERSTATE_RANGEFOGENABLE
D3DRENDERSTATE_SHADEMODE
D3DRENDERSTATE_SPECULARENABLE
D3DRENDERSTATE_SPECULARMATERIALSOURCE
D3DRENDERSTATE_VERTEXBLEND

The values set for the following texture-stage states:

D3DTSS_TEXCOORDINDEX
D3DTSS_TEXTURETRANSFORMFLAGS

- **Pixel-related States (D3DSBT_PIXEL)**

The values set for the following render states:

D3DRENDERSTATE_ALPHABLENDENABLE
D3DRENDERSTATE_ALPHAFUNC
D3DRENDERSTATE_ALPHAREF
D3DRENDERSTATE_ALPHATESTENABLE
D3DRENDERSTATE_ANTI_ALIAS

D3DRENDERSTATE_COLORKEYENABLE
D3DRENDERSTATE_DESTBLEND
D3DRENDERSTATE_DITHERENABLE
D3DRENDERSTATE_EDGEANTIALIAS
D3DRENDERSTATE_FILLMODE
D3DRENDERSTATE_FOGDENSITY
D3DRENDERSTATE_FOGEND
D3DRENDERSTATE_FOGSTART
D3DRENDERSTATE_LASTPIXEL
D3DRENDERSTATE_LINEPATTERN
D3DRENDERSTATE_SHADEMODE
D3DRENDERSTATE_SRCBLEND
D3DRENDERSTATE_STENCILENABLE
D3DRENDERSTATE_STENCILFAIL
D3DRENDERSTATE_STENCILFUNC
D3DRENDERSTATE_STENCILMASK
D3DRENDERSTATE_STENCILPASS
D3DRENDERSTATE_STENCILREF
D3DRENDERSTATE_STENCILWRITEMASK
D3DRENDERSTATE_STENCILZFAIL
D3DRENDERSTATE_STIPPLEDALPHA
D3DRENDERSTATE_TEXTUREFACTOR
D3DRENDERSTATE_TEXTUREPERSPECTIVE
D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7
D3DRENDERSTATE_ZBIAS
D3DRENDERSTATE_ZENABLE
D3DRENDERSTATE_ZFUNC
D3DRENDERSTATE_ZWRITEENABLE

The values set for the following texture-stage states:

D3DTSS_ADDRESS
D3DTSS_ADDRESSU
D3DTSS_ADDRESSV
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 **Direct3DDevice7.CreateStateBlock** method creates a new state block that contains the entire set of device states, or just 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 be captured into 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_VERTEX or D3DSBT_PIXEL). The following list summarizes the states that the system captures when you pass the D3DSBT_VERTEX or D3DSBT_PIXEL values:

- **Vertex-related States (D3DSBT_VERTEX)**

State (enabled or disabled) of all lights

Transformation matrices

User-defined clipping planes

The values set for the following render states:

D3DRENDERSTATE_AMBIENT

D3DRENDERSTATE_AMBIENTMATERIALSOURCE

D3DRENDERSTATE_CLIPPING

D3DRENDERSTATE_CLIPPLANEENABLE

D3DRENDERSTATE_COLORVERTEX

D3DRENDERSTATE_CULLMODE
D3DRENDERSTATE_DIFFUSEMATERIALSOURCE
D3DRENDERSTATE_EMISSIVEMATERIALSOURCE
D3DRENDERSTATE_EXTENTS
D3DRENDERSTATE_FOGCOLOR
D3DRENDERSTATE_FOGDENSITY
D3DRENDERSTATE_FOGENABLE
D3DRENDERSTATE_FOGEND
D3DRENDERSTATE_FOGSTART
D3DRENDERSTATE_FOGTABLEMODE
D3DRENDERSTATE_FOGVERTEXMODE
D3DRENDERSTATE_LIGHTING
D3DRENDERSTATE_LOCALVIEWER
D3DRENDERSTATE_NORMALIZENORMALS
D3DRENDERSTATE_RANGEFOGENABLE
D3DRENDERSTATE_SHADEMODE
D3DRENDERSTATE_SPECULARENABLE
D3DRENDERSTATE_SPECULARMATERIALSOURCE
D3DRENDERSTATE_VERTEXBLEND

The values set for the following texture-stage states:

D3DTSS_TEXCOORDINDEX
D3DTSS_TEXTURETRANSFORMFLAGS

- **Pixel-related States (D3DSBT_PIXEL)**

The values set for the following render states:

D3DRENDERSTATE_ALPHABLENDENABLE
D3DRENDERSTATE_ALPHAFUNC
D3DRENDERSTATE_ALPHAREF
D3DRENDERSTATE_ALPHATESTENABLE
D3DRENDERSTATE_ANTI_ALIAS
D3DRENDERSTATE_COLORKEYENABLE
D3DRENDERSTATE_DESTBLEND
D3DRENDERSTATE_DITHERENABLE
D3DRENDERSTATE_EDGEANTIALIAS
D3DRENDERSTATE_FILLMODE
D3DRENDERSTATE_FOGDENSITY
D3DRENDERSTATE_FOGEND
D3DRENDERSTATE_FOGSTART
D3DRENDERSTATE_LASTPIXEL

D3DRENDERSTATE_LINEPATTERN
D3DRENDERSTATE_SHADEMODE
D3DRENDERSTATE_SRCBLEND
D3DRENDERSTATE_STENCILENABLE
D3DRENDERSTATE_STENCILFAIL
D3DRENDERSTATE_STENCILFUNC
D3DRENDERSTATE_STENCILMASK
D3DRENDERSTATE_STENCILPASS
D3DRENDERSTATE_STENCILREF
D3DRENDERSTATE_STENCILWRITEMASK
D3DRENDERSTATE_STENCILZFAIL
D3DRENDERSTATE_STIPPLEDALPHA
D3DRENDERSTATE_TEXTUREFACTOR
D3DRENDERSTATE_TEXTUREPERSPECTIVE
D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7
D3DRENDERSTATE_ZBIAS
D3DRENDERSTATE_ZENABLE
D3DRENDERSTATE_ZFUNC
D3DRENDERSTATE_ZWRITEENABLE

The values set for the following texture-stage states:

D3DTSS_ADDRESS
D3DTSS_ADDRESSU
D3DTSS_ADDRESSV
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 it fails, it is most likely because the display mode has changed (error code DDERR_SURFACELOST). Your application should recover from this type of failure by recreating its surfaces, then recreating the state block.

Deleting State Blocks

[C++]

The **IDirect3DDevice7::DeleteStateBlock** deletes an existing state block, deallocating the memory used to contain it. The **DeleteStateBlock** method accepts the handle to the state block to be deleted at its only parameter. After deleting a state block, any handles to it are considered invalid, and can no longer be used.

[Visual Basic]

The **Direct3DDevice7.DeleteStateBlock** deletes an existing state block, deallocating the memory used to contain it. The **DeleteStateBlock** method accepts the handle to the state block to be deleted at its only parameter. After deleting a state block, any handles to it are considered invalid, and can no longer be used.

Using Devices

This section provides information about using Direct3D devices in a Direct3D Immediate Mode application. Information is divided into the following topics:

- Enumerating Direct3D Devices
- Creating a Direct3D Device
- Setting Transformations
- Rendering

Enumerating Direct3D Devices

Applications can query the hardware that they run on to detect which types of Direct3D devices it supports. This section contains information on two primary tasks involved in enumerating Direct3D Devices. Information is organized into the following topics:

- Starting Device Enumeration
- Selecting an Enumerated Device

Starting Device Enumeration

[C++]

A C++ application begins device enumeration by calling the **IDirect3D7::EnumDevices** method, which enumerates all the Direct3D devices that the hardware supports. Direct3D invokes the **D3DEnumDevicesCallback7** function to select the device to be used. You supply the **D3DEnumDevicesCallback7** function in your application. Because this callback function is supplied by your application, you can call it anything that you want.

The following code fragment illustrates the process of enumerating Direct3D devices. In this example, the device enumeration callback function is named EnumDeviceCallback. The code passes a pointer to EnumDeviceCallback to the **IDirect3D7::EnumDevices** method, which in turn calls EnumDeviceCallback for each device being enumerated, then returns. The callback can stop the enumeration before all devices have been enumerated by returning D3DENUMRET_CANCEL.

```
// In this code fragment, the variable lpd3d contains a valid
// pointer to the IDirect3D7 interface that the application obtained
// prior to executing this code.

BOOL fDeviceFound = FALSE;
hRes = lpd3d->EnumDevices(EnumDeviceCallback, &fDeviceFound);
if (FAILED(hRes))
{
    // Code to handle the error goes here.
}

if (!fDeviceFound)
{
    // Code to handle the error goes here.
}
```

[Visual Basic]

A Visual Basic application begins device enumeration by retrieving a reference to the **Direct3DEnumDevices** class by calling the **Direct3D7.GetDevicesEnum** method.

The **Direct3DEnumDevices** class defines methods that retrieve the number of enumerated devices and information about each device.

The following code fragment illustrates the process of retrieving the **Direct3DEnumDevices** class:

```
' In this example, the d3d variable is a valid reference to a
' Direct3D7 class.
Dim d3dEnum As Direct3DEnumDevices

' Retrieve the enumerator class.
Set d3dEnum = d3d.GetDevicesEnum
```

Selecting an Enumerated Device

[C++]

In C++, Direct3D invokes the **D3DEnumDevicesCallback7** function provided by the caller for each Direct3D device installed on the system. When it is called, its first and second parameters are strings that describe the device and provide the user-friendly device name.

The third parameter is a pointer to a **D3DDEVICEDESC7** structure containing information about the hardware capabilities of the device. Even if the device being enumerated is a HAL device, the particular hardware may not support all the capabilities that Direct3D allows. The last parameter is a programmer-defined value. Your application passes this value to the **IDirect3D7::EnumDevices** method. It in turn passes this value to the **D3DEnumDevicesCallback7** function.

The following code fragment illustrates how to create a **D3DEnumDevicesCallback7** function. In this example, the application-supplied callback function is named **EnumDeviceCallback**. The **EnumDeviceCallback** function uses the following algorithm to choose an appropriate Direct3D device:

1. Discard any devices that don't match the current display depth.
2. Discard any devices that can't do Gouraud-shaded triangles.
3. If a hardware device is found that passes the tests in points 1 and 2, use it. However, if the application is running in debug mode, it does not use the hardware device.

The code for the **EnumDeviceCallback** function is shown in the following example:

```
// This function is written with the assumption that the following
// global variables are declared in the program.
// DWORD          g_dwDeviceBitDepth      = 0;
// GUID           g_guidDevice;
// char           g_szDeviceName[MAX_PATH];
// char           g_szDeviceDesc[MAX_PATH];
// D3DDEVICEDESC7 g_d3dDeviceDesc;
```

```
static HRESULT WINAPI
EnumDeviceCallback(LPSTR      lpszDeviceDesc,
                  LPSTR      lpszDeviceName,
                  LPD3DDEVICEDESC7 lpd3dEnumDeviceDesc,
                  LPVOID      lpUserArg)
{
    BOOL flsHardware = FALSE;

    flsHardware = (lpd3dEnumDeviceDesc->dwDevCaps &
D3DDEVCAPS_HWRASTERIZATION);

    // Does the device render at the depth we want?
    if ((lpd3dEnumDeviceDesc->dwDeviceRenderBitDepth & g_dwDeviceBitDepth) == 0)
    {
        // If not, skip this device.

        return D3DENUMRET_OK;
    }

    // The device must support Gouraud-shaded triangles.
    if (!(lpd3dEnumDeviceDesc->dpcTriCaps.dwShadeCaps &
D3DPSHADECAPS_COLORGOURAUDRGB))
    {
        // No Gouraud shading. Skip this device.

        return D3DENUMRET_OK;
    }

    //
    // This device has the necessary capabilities. Save the details.
    //
    *(BOOL*)lpUserArg = TRUE;
    CopyMemory(&g_guidDevice, &lpd3dEnumDeviceDesc->deviceGUID, sizeof(GUID));
    strcpy(g_szDeviceDesc, lpszDeviceDesc);
    strcpy(g_szDeviceName, lpszDeviceName);
    CopyMemory(&g_d3dDeviceDesc, lpd3dEnumDeviceDesc,
        sizeof(D3DDEVICEDESC7));

    // If this is a hardware device, it has
    // the necessary capabilities.
    if (flsHardware)
        return D3DENUMRET_CANCEL;

    // Otherwise, keep looking.
```

```
    return D3DENUMRET_OK;
}
```

[Visual Basic]

In Visual Basic, the **Direct3DEnumDevices** class enumerates the Direct3D devices installed on the system. Retrieve information about each device by querying the class, identifying devices enumerated within the class by their index. The first device is at index 1, the second is at 2, and so on. The **Direct3DEnumDevices.GetCount** method reports the number of enumerated devices and, therefore, the highest allowable index value.

Call the **Direct3DEnumDevices.GetGuid** method to retrieve the GUID for an enumerated device as a text string. The string differs from one device to another.

Call the **Direct3DEnumDevices.GetName** and **Direct3DEnumDevices.GetDescription** methods to retrieve text strings that contain the name and user-friendly description of the device.

Call the **Direct3DEnumDevices.GetDesc** methods to retrieve a description of the capabilities of the device, in the form of the **D3DDEVICEDESC7** type. Even if the device being enumerated is a hardware device, the particular hardware might not support all of the capabilities that the Direct3D API allows.

The following code fragment illustrates how to create a function that calls the methods of the **Direct3DEnumDevices** class to determine device capabilities. In this example, the application-defined function uses the following algorithm to choose an appropriate Direct3D device:

1. Discard any devices that don't match the current display depth.
2. Discard any devices that can't do Gouraud-shaded triangles.
3. If a hardware device is found that passes the tests in points 1 and 2, use it.

```
Private Sub EnumerateDevices(d3denum As Direct3DEnumDevices)
```

```
' For this example, the following global variables are assumed to be defined:
```

```
  ' g_IDeviceBitDepth is the desired bit depth (combination of CONST_DDBITDEPTHFLAGS constants).
```

```
  ' g_DeviceDesc contains the D3DDEVICEDESC7 for device capabilities.
```

```
  ' g_strDeviceGUID contains the GUID string for the device.
```

```
  ' g_strDeviceDescription contains the string for the device description text.
```

```
  ' g_strDeviceName contains the string for the device name.
```

```
' Local variables used to enumerate devices.
```

```
Dim iDevice As Integer
```

```
Dim DevDesc As D3DDEVICEDESC7
```

```
Dim blsHardware As Boolean
```

```
' Begin enumerating the devices.
```

```
For iDevice = 1 To d3denum.GetCount
```

```
Dim checkDesc As D3DDEVICEDESC7
Call d3denum.GetDesc(iDevice, DevDesc)

' If there is no hardware support for this device
blsHardware = (DevDesc.IDevCaps Or D3DDEVCAPS_HWRASTERIZATION)

' Does the device render at the depth you want? If not, stop
' checking, and get the next device.
If checkDesc.IDeviceRenderBitDepth And Not g_IDeviceBitDepth Then
    GoTo NEXT_DEVICE
End If

' The device must support Gouraud-shaded triangles.
If (Not checkDesc.dpcTriCaps.IShadeCaps And
D3DPSHADECAPS_COLORGOURAUDRGB) Then
    ' No Gouraud shading in RGB mode. Skip this device.
    GoTo NEXT_DEVICE
End If

'
' By the time you get here, you know that this is a device you
' are interested in. Place the results in the global variables.
'

g_strDeviceGUID = d3denum.GetGuid(iDevice)
g_strDeviceDescription = d3denum.GetDescription(iDevice)
g_strDeviceName = d3denum.GetName(iDevice)
g_DeviceDesc = DevDesc

' If this current device is hardware accelerated,
' you have found what you are looking for.
If blsHardware = True Then Exit For

' Otherwise, keep looking.
NEXT_DEVICE:
Next iDevice
End Sub ' EnumerateDevices
```

Creating a Direct3D Device

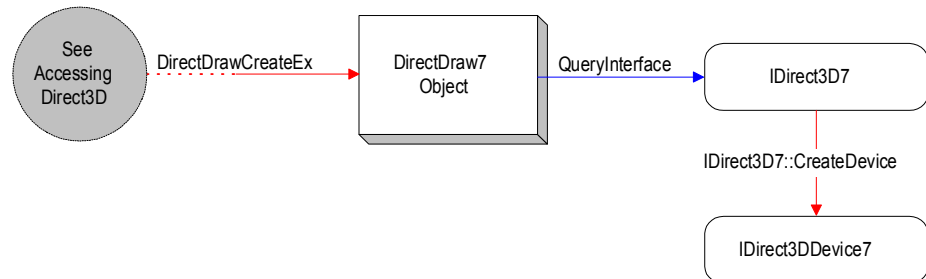
Important Performance Note

All rendering devices created by a given 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.

[C++]

To create a Direct3D device in a C++ application, your application must first create a DirectDraw object by calling the **DirectDrawCreateEx** function and obtain a pointer to the **IDirect3D7** interface. For details, see Accessing Direct3D. It must then call the **IDirect3D7::CreateDevice** method to create a Direct3D device. This method passes a pointer to the **IDirect3DDevice7** interface to your application.

The following illustration shows the process for creating a Direct3D device in C++.

**Note**

As discussed in Accessing Direct3D, the DirectDraw component comprises two COM objects. The newest object—called DirectDraw7 and created by calling the **DirectDrawCreateEx** function—is the only object that exposes the **IDirect3D7** interface. It is the only object capable of spawning a Direct3D device that exposes the **IDirect3DDevice7** interface. The Direct3DDevice object created in this manner does not expose legacy Direct3D interfaces. Applications that require previous iterations of the Direct3D device interface must use the **DirectDrawCreate** method to create a DirectDraw object, and then create the desired legacy objects.

This code fragment illustrates the process of creating a Direct3D rendering device that supports the **IDirect3DDevice7** interface:

```

LPDIRECTDRAW7 lpDD;          // IDirectDraw7 Interface
LPDIRECT3D7 lpD3D;          // IDirect3D7 Interface
LPDIRECTDRAW7SURFACE7 lpddsRender; // Rendering surface
LPDIRECT3DDEVICE7 lpD3DDevice; // D3D Device
HRESULT hr;

// Create DirectDraw interface.
// Use the current display driver.
hr = DirectDrawCreateEx(NULL, (void **)&lpDD, IID_IDirectDraw7, NULL);
if (FAILED(hr))
{
    // Code to handle the error goes here.
}

// Get an IDirect3D7 interface
  
```

```

hr =
    lpDD->QueryInterface (IID_IDirect3D7, (void **)&lpD3D);
if (FAILED (hr))
{
    // Code to handle the error goes here.
}

//
// Code for the following tasks is omitted for clarity.
//
// Applications must set the cooperative level at this point.
// Full-screen applications probably need to set the display
// mode.
// The primary surface should be created here.
// The rendering surface must be created at this point. It is
// assumed in this code fragment that, once created, the rendering
// surface is pointed to by the variable lpddsRender.
// If a z-buffer is being used, it should be created here.
// Direct3D device enumeration can be done at this point.

hr = lpD3D->CreateDevice (IID_IDirect3DHALDevice,
                        lpddsRender,
                        &lpd3dDevice,
                        NULL);

```

The preceding sample invokes the **IDirect3D7::CreateDevice** method to create the Direct3D device. In the case of this sample, a Direct3D HAL device is created if the call is successful.

The target DirectDraw rendering surface that your application creates must be created to be used as a Direct3D rendering target. To do this, it must pass a **DDSURFACEDESC2** structure to the **IDirectDraw7::CreateSurface** method. The **DDSURFACEDESC2** structure has a member called **ddsCaps**, which is a structure of type **DDSCAPS2**. The **DDSCAPS2** structure contains a member named **dwCaps**, which must be set to **DDSCAPS_3DDEVICE** when **IDirectDraw7::CreateSurface** is invoked.

When it is used with a hardware-accelerated rendering device, the render-target surface that you use must be created in display memory (with the **DDSCAPS_VIDEOMEMORY** flag). Otherwise, it must be created in system memory (using the **DDSCAPS_SYSTEMMEMORY** flag).

Note

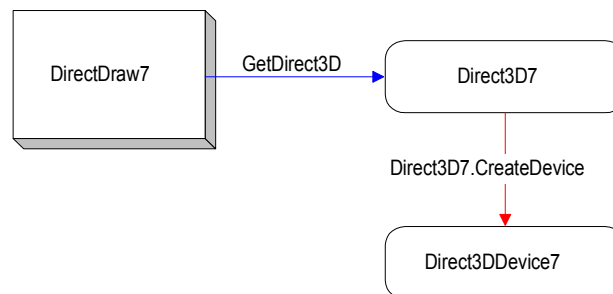
Some popular hardware devices require that the render target and depth buffer surfaces use the same bit depth. On such hardware, if your application uses a 16-bit render-target surface, the attached depth buffer must also be 16 bits. For a 32-bit render-target surface, the depth buffer must be 32 bits, of which 8 bits can be used for stencil buffering (if needed).

If the hardware upon which your application is running has this requirement and your application fails to meet it, any attempts to create a rendering device that uses the noncompliant surfaces fail. Use the DirectDraw method, **IDirectDraw7::GetDeviceIdentifier** to track hardware that imposes this limitation.

[Visual Basic]

To create a Direct3D device, your application must first initialize the DirectDraw object in the normal manner and obtain a reference to the **Direct3D7** class. For details, see Accessing Direct3D. It must then call the **Direct3D7.CreateDevice** method to create a Direct3D device. The method returns a reference to a **Direct3DDevice7** class object.

The following figure illustrates the process for creating a Direct3D device from a Visual Basic application:



The following code fragment illustrates this process:

```

' For this example, the g_dx variable contains a valid
' reference to a DirectX7 object.
Private Sub Test()
    On Local Error Resume Next

    Dim ddraw As DirectDraw7      ' DirectDraw7 class
    Dim d3d As Direct3D7         ' Direct3D3 class
    Dim ddsRender As DirectDrawSurface7 ' Rendering surface
    Dim d3dDev As Direct3DDevice7 ' Direct3D Device

    ' Create DirectDraw7 object.
    ' Use the current display driver.
    Set ddraw = g_dx.DirectDrawCreate("")
    If Err.Number <> DD_OK Then
        ' Code to handle the error goes here.
    End If

    ' Get a Direct3D7 object.
  
```

```
Set d3d = ddraw.GetDirect3D
If Err.Number <> DD_OK Then
    ' Code to handle the error goes here.
End If

'
' Code for the following tasks is omitted for clarity.
'
' + Applications need to set the cooperative level at this point.
' + Full-screen applications probably need to set the display
'   mode.
' + The primary surface should be created here.
' + The rendering surface must be created at this point. It is
'   assumed in this code fragment that, once created, the
'   rendering surface is referred to by the variable ddsRender.
' + If a depth-buffer is being used, it should be created here.
' + Direct3D device enumeration can be done at this point.

Set d3dDev = d3d.CreateDevice("IID_IDirect3DHALDevice", ddsRender)
End Sub
```

This code fragment invokes the **Direct3D7.CreateDevice** method to create the Direct3D device. In this case, a Direct3D HAL device is created if the call is successful.

The target DirectDraw rendering surface that your application creates must be created for use as a Direct3D rendering target. To do this, it must pass a **DDSURFACEDESC2** variable to the **DirectDraw7.CreateSurface** method. The **DDSURFACEDESC2** type has a member called **ddsCaps**, which is of type **DDSCAPS2**. The **DDSCAPS2** type contains a member named **ICaps**, which must be set to **DDSCAPS_3DDEVICE** when **DirectDraw7.CreateSurface** is invoked.

To be used with a hardware-accelerated rendering device, the render-target surface that you use must be created in display memory (with the **DDSCAPS_VIDEOMEMORY** flag). Otherwise, it must be created in system memory (using the **DDSCAPS_SYSTEMMEMORY** flag).

Note

Some popular hardware devices require that the render target and depth buffer surfaces use the same bit depth. On such hardware, if your application uses a 16-bit render-target surface, the attached depth buffer must also be 16 bits. For a 32-bit render-target surface, the depth buffer must be 32 bits, of which 8 bits can be used for stencil buffering, if needed.

If the hardware upon which your application is running has this requirement and your application fails to meet it, any attempts to create a rendering device that uses the noncompliant surfaces fail.

Setting Transformations

[C++]

In C++, transformations are applied by using the **IDirect3DDevice7::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 = lpDev->SetTransform(D3DTRANSFORMSTATE_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 **IDirect3DDevice7::SetTransform**. The most common are **D3DTRANSFORMSTATE_WORLD**, **D3DTRANSFORMSTATE_VIEW**, and **D3DTRANSFORMSTATE_PROJECTION**. These transformation states are defined in the **D3DTRANSFORMSTATETYPE** enumerated type. This enumeration also contains values to set transformations to be applied to texture coordinates: the **D3DTRANSFORMSTATE_TEXTURE1** through **D3DTRANSFORMSTATE_TEXTURE7** states.

[Visual Basic]

In Visual Basic, transformations are applied by using the **Direct3DDevice7.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 d3dDev.SetTransform(D3DTRANSFORMSTATE_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 **Direct3DDevice7.SetTransform**. The most common are **D3DTRANSFORMSTATE_WORLD**, **D3DTRANSFORMSTATE_VIEW**, and **D3DTRANSFORMSTATE_PROJECTION**. These transformation states are defined in the **CONST_D3DTRANSFORMSTATETYPE** enumeration. This enumeration also contains values to set transformations to be applied to texture coordinates: the

D3DTRANSFORMSTATE_TEXTURE1 through
D3DTRANSFORMSTATE_TEXTURE7 states.

Rendering

Applications use the DrawPrimitive family of methods to render a 3-D scene. The DrawPrimitive architecture is recommended over the legacy execute-buffer rendering model. As a result, the following topics discuss rendering with DrawPrimitive:

- Grouping Vertices
- Beginning and Ending a Scene
- Clearing Surfaces
- Rendering Primitives
- Primitive Types
- Render States

Grouping Vertices

You can group vertices for a primitive in two basic ways:

- In an array that includes all vertices for the primitive in order.
- In an unordered array of vertices, accompanied by an ordered array of values, where each value is the index of a vertex in the unordered array. Primitives of this type are sometimes referred to as indexed primitives.

The DrawPrimitive architecture makes it possible to render groups of vertices that define a primitive in a single call or to render the vertices that comprise a primitive one by one. For more information, see Rendering Primitives.

Beginning and Ending a Scene

[C++]

Applications written in C++ notify Direct3D that scene rendering is about to begin by calling the **IDirect3DDevice7::BeginScene** method. **BeginScene** causes the system to check its internal data structures, the availability and validity of rendering surfaces, and 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, returning D3DERR_SCENE_NOT_IN_SCENE. For more information, see Rendering Primitives.

After you complete the scene, call the **IDirect3DDevice7::EndScene** method. The **EndScene** method clears an internal flag that signals when a scene is in progress, flushes cached data, and verifies the integrity of the rendering surfaces.

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. When surfaces are lost during rendering, **EndScene** will return DDERR_SURFACELOST. If surfaces were not successfully restored before calling **BeginScene**, then **BeginScene** will also return DDERR_SURFACELOST.

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 of 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 this:

```
HRESULT hr;

if(SUCCEEDED(lpDevice->BeginScene()))
{
    // Render primitives only if the scene
    // starts successfully.

    // Close the scene.
    hr = lpDevice->EndScene();
    if(FAILED(hr))
        return hr;
}
```

You cannot embed scenes; that is, you must complete rendering a scene before you can begin another one. Calling **IDirect3DDevice7::EndScene** when **IDirect3DDevice7::BeginScene** has not been called returns the D3DERR_SCENE_NOT_IN_SCENE error value. Likewise, calling **BeginScene** when a previous scene has not been completed (with the **EndScene** method) results in the D3DERR_SCENE_IN_SCENE error.

Do not attempt to call GDI functions on DirectDraw surfaces, such as the render target or textures, while a scene is being rendered (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.

[\[Visual Basic\]](#)

Visual Basic applications notify Direct3D that scene rendering is about to begin by calling the **Direct3DDevice7.BeginScene** method. **BeginScene** causes the system to check its internal data structures, the availability and validity of rendering surfaces,

and 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, returning the D3DERR_SCENE_NOT_IN_SCENE error. For more information, see *Rendering Primitives*.

After you complete the scene, call the **Direct3DDevice7.EndScene** method. The **EndScene** method clears an internal flag that signals when a scene is in progress, flushes cached data, and verifies the integrity of the rendering surfaces.

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** will be 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, raising the DDERR_SURFACELOST error. If surfaces were not successfully restored before calling **BeginScene**, then **BeginScene** will also return DDERR_SURFACELOST.

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 of 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 this:

On Local Error Resume Next

```
Call d3dDev.BeginScene
If Err.Number = DD_OK Then
    ' Render primitives only if the scene
    ' starts successfully.

    ' Close the scene.
    Call d3dDev.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 **Direct3DDevice7.EndScene** when **Direct3DDevice7.BeginScene** has not been called causes a D3DERR_SCENE_NOT_IN_SCENE error. Likewise, calling **BeginScene** when a previous scene has not been completed (with the **EndScene** method) results in the D3DERR_SCENE_IN_SCENE error.

Do not attempt to call GDI functions on DirectDraw surfaces, such as the render target or textures, while a scene is being rendered (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, make sure that all GDI calls are made outside the scene functions.

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 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++]

The **IDirect3DDevice7** interface offers the **IDirect3DDevice7::Clear** method to clear the viewport. For more information about using these methods, see [Clearing a Viewport](#) in the Viewports and Clipping section.

[Visual Basic]

The **Direct3DDevice7** Visual Basic class offers the **Direct3DDevice7.Clear** method to clear the viewport. For more information about using these methods, see [Clearing a Viewport](#) in the Viewports and Clipping section.

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 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.

Rendering Primitives

The following topics introduce the DrawPrimitive rendering methods and provides information about using them in your application:

- [DrawPrimitive Methods](#)
- [Rendering Strided Vertices](#)

DrawPrimitive Methods

[C++]

Direct3D offers several methods to render primitives and indexed primitives in a single call. All rendering methods in the **IDirect3DDevice7** interface accept a combination of flexible vertex format flags that describe the vertex type your application uses; these flags also determine which parts of the geometry pipeline the system will apply. (This differs from the version of the methods in the **IDirect3DDevice2**, which accept only the concrete vertex formats identified by members of the **D3DPRIMITIVETYPE** enumeration.) For more information about these descriptors, see Vertex Formats.

The DrawPrimitive family of rendering methods offers the following rendering methods, organized into the following groups:

Basic rendering

IDirect3DDevice7::DrawPrimitive and
IDirect3DDevice7::DrawIndexedPrimitive

Vertex buffer rendering

IDirect3DDevice7::DrawPrimitiveVB and
IDirect3DDevice7::DrawIndexedPrimitiveVB

Strided vertices

IDirect3DDevice7::DrawPrimitiveStrided and
IDirect3DDevice7::DrawIndexedPrimitiveStrided. Vertex buffers can be used to render strided vertices, but the vertices must be processed with the **IDirect3DVertexBuffer7::ProcessVerticesStrided** method.

When you render a primitive, the associated methods accept parameters that describe the type of primitive being rendered (such as a triangle strip, a triangle list, or another primitive type), the vertex format, rendering behavior flags, and vertex information. These flags and their effects are documented in the references for the rendering methods.

Note that the number of vertices you need to provide for the methods depends on the type of primitive you are rendering. For instance, if you're rendering a line list, you must provide at least two vertices to define a single line, and the total number must be an even value. Likewise for a triangle list, you must include at least three vertices, with the total evenly divisible by three. For information about vertex counts for other types of primitives, see Primitive Types and **D3DPRIMITIVETYPE**.

Vertex buffers, and the methods to render from them, provide performance and ease-of-use enhancements that improve upon rendering vertices from your own data structures. For more information, see Vertex Buffers.

[Visual Basic]

Direct3D offers Visual Basic applications several methods to render primitives in a single call. All rendering methods in the **Direct3DDevice7** class accept a combination of flexible vertex format flags that describe the vertex type your application uses;

these flags also determine which parts of the geometry pipeline the system will apply. For more information about these descriptors, see [Vertex Formats](#).

The DrawPrimitive family of rendering methods offers the following rendering methods, organized into the following groups:

Basic rendering

Direct3DDevice7.DrawPrimitive and
Direct3DDevice7.DrawIndexedPrimitive

Vertex buffer rendering

Direct3DDevice7.DrawPrimitiveVB and
Direct3DDevice7.DrawIndexedPrimitiveVB

When you render a primitive, the associated methods accept parameters that describe the type of primitive being rendered (such as a triangle strip, a triangle list, or another primitive type), the vertex format, rendering behavior flags, and vertex information. These flags and their effects are documented in the references for the rendering methods.

Note that the number of vertices you need to provide for the methods depends on the type of primitive you are rendering. For instance, if you're rendering a line list, you must provide at least two vertices to define a single line, and the total number must be an even value. Likewise for a triangle list, you must include at least three vertices, with the total evenly divisible by three. For information about vertex counts for other types of primitives, see [Primitive Types](#) and **CONST_D3DPRIMITIVETYPE**.

Vertex buffers, and the methods to render from them, provide performance and ease-of-use enhancements that improve upon rendering vertices from your own data structures. For more information, see [Vertex Buffers](#).

Note

DirectX for Visual Basic does not support rendering strided vertices.

Rendering Strided Vertices

[\[Visual Basic\]](#)

Note

The information in this topic applies only to applications written in C++. DirectX for Visual Basic does not support rendering with strided vertices.

[\[C++\]](#)

Because of the indirection provided by strided vertices—detailed in the [Strided Vertex Format](#) section—applications must take care to set up the vertices properly. Often, developers forget to include a vertex component in the associated **D3DDRAWPRIMITIVESTRIEDEDDATA** structures. This oversight doesn't

necessarily cause the rendering methods to fail, but can result in "missing" geometry that is difficult to troubleshoot.

The following code shows one way that you might set up and render strided vertices:

```
//-----
// A custom vertex format that includes XYZ, a
// diffuse color & two sets of texture coords.
//-----
struct MTVERTEX
{
    FLOAT x, y, z;
    DWORD dwColor;
    FLOAT tuBase, tvBase;
    FLOAT tuLightMap, tvLightMap;
};

// Make an array of custom vertices.
MTVERTEX g_avVertices[36];
// Fill the array.
// (vertex at index 0)
// .
// .
// .
// (vertex at index 35)

// Construct strided vertices vertex using the array of
// custom vertices already defined.
D3DDRAWPRIMITIVESTRIDEDDATA g_StridedData;

// Assign the addresses of the various interleaved components
// to their corresponding strided members.
g_StridedData.position.lpvData = &g_avWallVertices[24].x;
g_StridedData.diffuse.lpvData = &g_avWallVertices[24].dwColor;
g_StridedData.textureCoords[0].lpvData = &g_avWallVertices[24].tuBase;
g_StridedData.textureCoords[1].lpvData = &g_avWallVertices[24].tuLightMap;
g_StridedData.position.dwStride = sizeof(MTVERTEX);
g_StridedData.diffuse.dwStride = sizeof(MTVERTEX);
g_StridedData.textureCoords[0].dwStride = sizeof(MTVERTEX);
g_StridedData.textureCoords[1].dwStride = sizeof(MTVERTEX);

// Render the vertices with multiple texture blending (Modulate).
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_MODULATE );
g_pd3dDevice->SetTexture( 0, g_BaseTextureMap);
g_pd3dDevice->SetTexture( 1, g_LightMap);
g_pd3dDevice->DrawPrimitiveStrided( D3DPT_TRIANGLELIST,
                                   D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX2,
```

&g_StridedData, 12, NULL);

Primitive Types

Direct3D 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 these primitive types from a C++ application with any of the rendering methods of the **IDirect3DDevice7** interface. For more information, see Rendering. (Note that you cannot render point lists with the indexed-primitive rendering methods.)

[Visual Basic]

You can render all of these primitive types from a Visual Basic application with any of the rendering methods of the **Direct3DDevice7** interface. For more information, see Rendering.

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. Applications create a point list by filling an array of vertices.

[C++]

The following C++ code fragment illustrates the process of constructing a point list:

```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

lpVerts[0] = D3DVERTEX(D3DVECTOR(-5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[1] = D3DVERTEX(D3DVECTOR(0,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[2] = D3DVERTEX(D3DVECTOR(5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[3] = D3DVERTEX(D3DVECTOR(10,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[4] = D3DVERTEX(D3DVECTOR(15,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[5] = D3DVERTEX(D3DVECTOR(20,5,0),D3DVECTOR(0,0,-1),0,0);
```

Render the point list using the **IDirect3DDevice7::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice7::DrawPrimitive** for

drawing the point list in the preceding example. All calls to **IDirect3DDevice7::DrawPrimitive** must occur between **IDirect3DDevice7::BeginScene** and **IDirect3DDevice7::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice is a valid
// pointer to an IDirect3DDevice7 interface.
hResult =
    lpDirect3DDevice->DrawPrimitive(D3DPT_POINTLIST,
                                    D3DFVF_VERTEX,
                                    lpVerts,
                                    TOTAL_VERTS,
                                    D3DDP_WAIT);
```

[Visual Basic]

An application written in Visual Basic creates a point list by filling an array of vertices, as in the following code fragment:

```
' This code fragment assumes that dx is a valid reference
' to a DirectX7 object.
On Local Error Resume Next
Const TOTAL_VERTS As Integer = 6
Dim Verts(TOTAL_VERTS) As D3DVERTEX

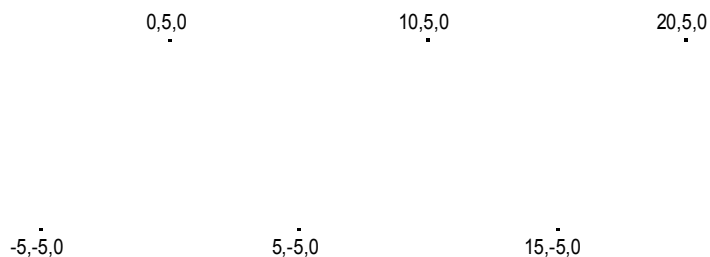
Verts(0) = dx.CreateD3DVertex(-5, -5, 0, 0, 0, -1, 0, 0)
Verts(1) = dx.CreateD3DVertex(0, 5, 0, 0, 0, -1, 0, 0)
Verts(2) = dx.CreateD3DVertex(5, -5, 0, 0, 0, -1, 0, 0)
Verts(3) = dx.CreateD3DVertex(10, 5, 0, 0, 0, -1, 0, 0)
Verts(4) = dx.CreateD3DVertex(15, -5, 0, 0, 0, -1, 0, 0)
Verts(5) = dx.CreateD3DVertex(20, 5, 0, 0, 0, -1, 0, 0)
```

Render the point list using the **Direct3DDevice7.DrawPrimitive** method. The following code fragment illustrates the use of **Direct3DDevice7.DrawPrimitive** for drawing the point list in the preceding example. All calls to **Direct3DDevice7.DrawPrimitive** must occur between **Direct3DDevice7.BeginScene** and **Direct3DDevice7.EndScene**.

```
' This code fragment assumes that d3ddev is a valid
' reference to an IDirect3DDevice7 object.
Call d3ddev.DrawPrimitive(D3DPT_POINTLIST, _
    D3DFVF_VERTEX, _
    Verts(0), _
    TOTAL_VERTS, _
    D3DDP_WAIT)

If Err.Number <> DD_OK Then
    ' Code to handle error goes here.
End If
```

The following illustration shows the resulting points.



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

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. The number of vertices in a line list must be greater than or equal to two, and it must be even.

[C++]

The following code fragment illustrates how this task is accomplished from C++:

```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

lpVerts[0] = D3DVERTEX(D3DVECTOR(-5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[1] = D3DVERTEX(D3DVECTOR(0,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[2] = D3DVERTEX(D3DVECTOR(5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[3] = D3DVERTEX(D3DVECTOR(10,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[4] = D3DVERTEX(D3DVECTOR(15,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[5] = D3DVERTEX(D3DVECTOR(20,5,0),D3DVECTOR(0,0,-1),0,0);
```

Render the line list using the **IDirect3DDevice7::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice7::DrawPrimitive** for drawing the line list in the preceding example. Remember that all calls to **IDirect3DDevice7::DrawPrimitive** must occur between **IDirect3DDevice7::BeginScene** and **IDirect3DDevice7::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice is a valid
// pointer to an IDirect3DDevice7 interface.
hResult =
    lpDirect3DDevice->DrawPrimitive(D3DPT_LINELIST,
                                    D3DFVF_VERTEX,
                                    lpVerts,
```

```
TOTAL_VERTS,
D3DDP_WAIT);
```

[Visual Basic]

A Visual Basic application could fill the array as shown in the following code fragment:

```
' This code fragment assumes that dx is a valid reference
' to a DirectX7 object.
On Local Error Resume Next
Const TOTAL_VERTS As Integer = 6
Dim Verts(TOTAL_VERTS) As D3DVERTEX
```

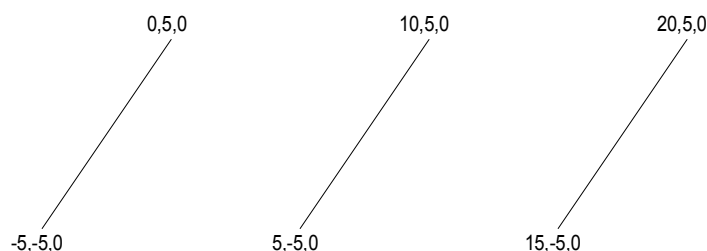
```
Verts(0) = dx.CreateD3DVertex(-5, -5, 0, 0, 0, -1, 0, 0)
Verts(1) = dx.CreateD3DVertex(0, 5, 0, 0, 0, -1, 0, 0)
Verts(2) = dx.CreateD3DVertex(5, -5, 0, 0, 0, -1, 0, 0)
Verts(3) = dx.CreateD3DVertex(10, 5, 0, 0, 0, -1, 0, 0)
Verts(4) = dx.CreateD3DVertex(15, -5, 0, 0, 0, -1, 0, 0)
Verts(5) = dx.CreateD3DVertex(20, 5, 0, 0, 0, -1, 0, 0)
```

Render the point list using the **Direct3DDevice7.DrawPrimitive** method. The following code fragment illustrates the use of **Direct3DDevice7.DrawPrimitive** for drawing the line list in the preceding example. All calls to **Direct3DDevice7.DrawPrimitive** must occur between **Direct3DDevice7.BeginScene** and **Direct3DDevice7.EndScene**.

```
' This code fragment assumes that d3ddev is a valid
' reference to an Direct3DDevice7 object.
Call d3ddev.DrawPrimitive(D3DPT_LINELIST, _
    D3DFVF_VERTEX, _
    Verts(0), _
    TOTAL_VERTS, _
    D3DDP_WAIT)

If Err.Number <> DD_OK Then
    ' Code to handle error goes here.
End If
```

The following illustration shows the resulting lines.



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

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. You create a line strip by filling an array of vertices.

[C++]

The following code fragment illustrates this process using C++:

```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

lpVerts[0] = D3DVERTEX(D3DVECTOR(-5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[1] = D3DVERTEX(D3DVECTOR(0,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[2] = D3DVERTEX(D3DVECTOR(5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[3] = D3DVERTEX(D3DVECTOR(10,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[4] = D3DVERTEX(D3DVECTOR(15,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[5] = D3DVERTEX(D3DVECTOR(20,5,0),D3DVECTOR(0,0,-1),0,0);
```

Render the line list using the **IDirect3DDevice7::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice7::DrawPrimitive** for drawing the line strip in the preceding example. Remember that all calls to **IDirect3DDevice7::DrawPrimitive** must occur between **IDirect3DDevice7::BeginScene** and **IDirect3DDevice7::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice is a valid
// pointer to an IDirect3DDevice7 interface.
hResult =
    lpDirect3DDevice->DrawPrimitive(D3DPT_LINESTRIP,
                                    D3DFVF_VERTEX,
                                    lpVerts,
                                    TOTAL_VERTS,
                                    D3DDP_WAIT);
```

[Visual Basic]

This code fragment illustrates this process using Visual Basic.

```
' This code fragment assumes that dx is a valid reference
' to a DirectX7 object.
On Local Error Resume Next
Const TOTAL_VERTS As Integer = 6
Dim Verts(TOTAL_VERTS) As D3DVERTEX
```

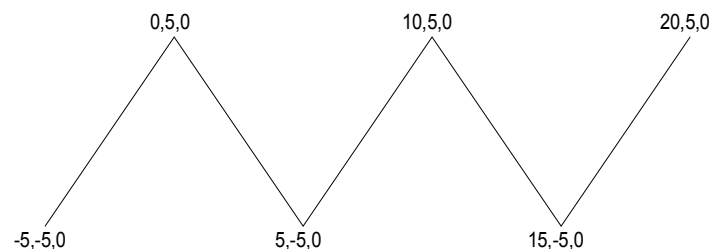
```
Verts(0) = dx.CreateD3DVertex(-5, -5, 0, 0, 0, -1, 0, 0)
Verts(1) = dx.CreateD3DVertex(0, 5, 0, 0, 0, -1, 0, 0)
Verts(2) = dx.CreateD3DVertex(5, -5, 0, 0, 0, -1, 0, 0)
Verts(3) = dx.CreateD3DVertex(10, 5, 0, 0, 0, -1, 0, 0)
Verts(4) = dx.CreateD3DVertex(15, -5, 0, 0, 0, -1, 0, 0)
Verts(5) = dx.CreateD3DVertex(20, 5, 0, 0, 0, -1, 0, 0)
```

Render the line list using the **Direct3DDevice7.DrawPrimitive** method. The following code fragment illustrates the use of **Direct3DDevice7.DrawPrimitive** for drawing the line strip in the preceding example. Remember that all calls to **Direct3DDevice7.DrawPrimitive** must occur between **Direct3DDevice7.BeginScene** and **Direct3DDevice7.EndScene**.

```
' This code fragment assumes that d3ddev is a valid
' reference to an Direct3DDevice7 object.
Call d3ddev.DrawPrimitive(D3DPT_LINESTRIP, _
    D3DFVF_VERTEX, _
    Verts(0), _
    TOTAL_VERTS, _
    D3DDP_WAIT)

If Err.Number <> DD_OK Then
    ' Code to handle error goes here.
End If
```

The following illustration shows the line strip that the previous code produces.



Triangle Lists

A triangle list is a list of isolated triangles. They may or may 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 when you want 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 to the triangle list that appears to emit light. Each triangle in the wall appears to glow. The scene behind the wall becomes partially visible through the gaps between the triangles, as players 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.

[C++]

Create a triangle list in C++ by filling an array of vertices, as in the following code fragment:

```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

lpVerts[0] = D3DVERTEX(D3DVECTOR(-5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[1] = D3DVERTEX(D3DVECTOR(0,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[2] = D3DVERTEX(D3DVECTOR(5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[3] = D3DVERTEX(D3DVECTOR(10,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[4] = D3DVERTEX(D3DVECTOR(15,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[5] = D3DVERTEX(D3DVECTOR(20,5,0),D3DVECTOR(0,0,-1),0,0);
```

Render a triangle list using the **IDirect3DDevice7::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice7::DrawPrimitive** for drawing the triangle list in the preceding example. Remember that all calls to **IDirect3DDevice7::DrawPrimitive** must occur between **IDirect3DDevice7::BeginScene** and **IDirect3DDevice7::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice is a valid
// pointer to an IDirect3DDevice7 interface.
hResult =
    lpDirect3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST,
                                    D3DFVF_VERTEX,
                                    lpVerts,
                                    TOTAL_VERTS,
                                    D3DDP_WAIT);
```

[Visual Basic]

Create a triangle list in Visual Basic by filling an array of vertices, as in the following code fragment:

```
' This code fragment assumes that dx is a valid reference
' to a DirectX7 object.
On Local Error Resume Next
Const TOTAL_VERTS As Integer = 6
Dim Verts(TOTAL_VERTS) As D3DVERTEX
```

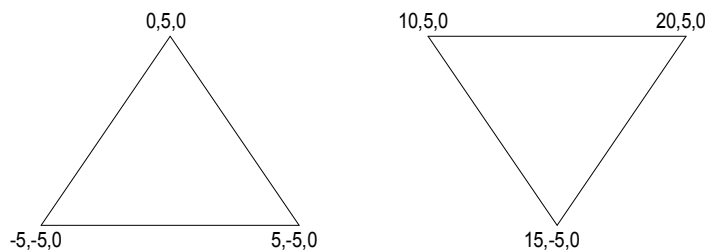
```
Verts(0) = dx.CreateD3DVertex(-5, -5, 0, 0, 0, -1, 0, 0)
Verts(1) = dx.CreateD3DVertex(0, 5, 0, 0, 0, -1, 0, 0)
Verts(2) = dx.CreateD3DVertex(5, -5, 0, 0, 0, -1, 0, 0)
Verts(3) = dx.CreateD3DVertex(10, 5, 0, 0, 0, -1, 0, 0)
Verts(4) = dx.CreateD3DVertex(15, -5, 0, 0, 0, -1, 0, 0)
Verts(5) = dx.CreateD3DVertex(20, 5, 0, 0, 0, -1, 0, 0)
```

Render a triangle list using the **Direct3DDevice7.DrawPrimitive** method. The following code fragment illustrates the use of **Direct3DDevice7.DrawPrimitive** for drawing the triangle list in the preceding example. Remember that all calls to **Direct3DDevice7.DrawPrimitive** must occur between **Direct3DDevice7.BeginScene** and **Direct3DDevice7.EndScene**.

```
' This code fragment assumes that d3ddev is a valid
' reference to an Direct3DDevice7 object.
Call d3ddev.DrawPrimitive(D3DPT_TRIANGLELIST, _
    D3DFVF_VERTEX, _
    Verts(0), _
    TOTAL_VERTS, _
    D3DDP_WAIT)

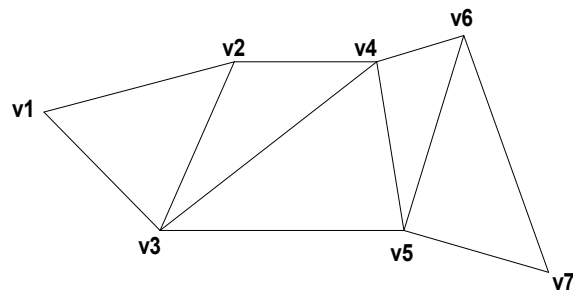
If Err.Number <> DD_OK Then
    ' Code to handle error goes here.
End If
```

The following illustration depicts the resulting triangles.



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 only need 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 of 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. You create a triangle strip by filling an array of vertices.

[C++]

The following C++ code fragment fills an array of vertices for use as a triangle strip:

```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

lpVerts[0] = D3DVERTEX(D3DVECTOR(-5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[1] = D3DVERTEX(D3DVECTOR(0,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[2] = D3DVERTEX(D3DVECTOR(5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[3] = D3DVERTEX(D3DVECTOR(10,5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[4] = D3DVERTEX(D3DVECTOR(15,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[5] = D3DVERTEX(D3DVECTOR(20,5,0),D3DVECTOR(0,0,-1),0,0);
```

Your application can then render the triangle strip using the

IDirect3DDevice7::DrawPrimitive method. The following code fragment illustrates the use of **IDirect3DDevice7::DrawPrimitive** for drawing the triangle strip in the preceding example. Remember that all calls to **IDirect3DDevice7::DrawPrimitive** must occur between **IDirect3DDevice7::BeginScene** and **IDirect3DDevice7::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice7 is a valid
// pointer to an IDirect3DDevice7 interface.
hResult =
    lpDirect3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP,
                                    D3DFVF_VERTEX,
                                    lpVerts,
                                    TOTAL_VERTS,
```

D3DDP_WAIT);

[Visual Basic]

Applications written in Visual Basic could use code like the following to setup a triangle strip:

```
' This code fragment assumes that dx is a valid reference
' to a DirectX7 object.
On Local Error Resume Next
Const TOTAL_VERTS As Integer = 6
Dim Verts(TOTAL_VERTS) As D3DVERTEX
```

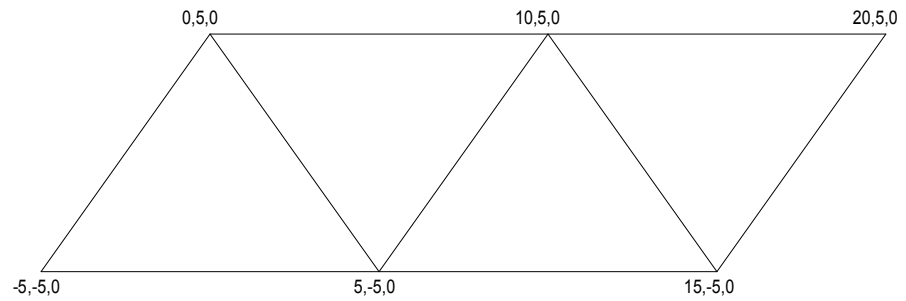
```
Verts(0) = dx.CreateD3DVertex(-5, -5, 0, 0, 0, -1, 0, 0)
Verts(1) = dx.CreateD3DVertex(0, 5, 0, 0, 0, -1, 0, 0)
Verts(2) = dx.CreateD3DVertex(5, -5, 0, 0, 0, -1, 0, 0)
Verts(3) = dx.CreateD3DVertex(10, 5, 0, 0, 0, -1, 0, 0)
Verts(4) = dx.CreateD3DVertex(15, -5, 0, 0, 0, -1, 0, 0)
Verts(5) = dx.CreateD3DVertex(20, 5, 0, 0, 0, -1, 0, 0)
```

Your application can then render the triangle strip using the **Direct3DDevice7.DrawPrimitive** method. The following code fragment illustrates the use of **Direct3DDevice7.DrawPrimitive** for drawing the triangle strip in the preceding example. Remember that all calls to **Direct3DDevice7.DrawPrimitive** must occur between **Direct3DDevice7.BeginScene** and **Direct3DDevice7.EndScene**.

```
' This code fragment assumes that d3ddev is a valid
' reference to an Direct3DDevice7 object.
Call d3ddev.DrawPrimitive(D3DPT_TRIANGLESTRIP, _
    D3DFVF_VERTEX, _
    Verts(0), _
    TOTAL_VERTS, _
    D3DDP_WAIT)

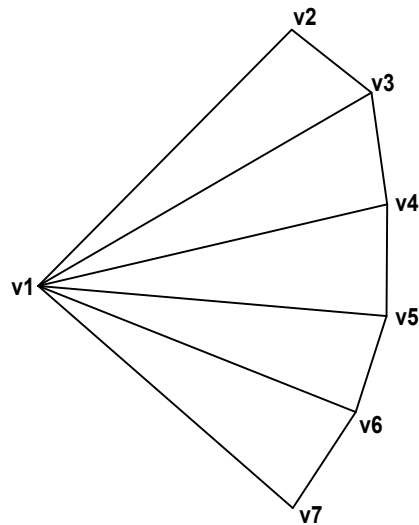
If Err.Number <> DD_OK Then
    ' Code to handle error goes here.
End If
```

The following illustration shows the resulting triangle strip.



Triangle Fans

A triangle fan is similar to a triangle strip, except that all of 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.)

[C++]

Your C++ application can create a triangle fan by filling an array of vertices, as shown in the following code fragment:

```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

lpVerts[0] = D3DVERTEX(D3DVECTOR(0,0,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[1] = D3DVERTEX(D3DVECTOR(-5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[2] = D3DVERTEX(D3DVECTOR(-3,7,0),D3DVECTOR(0,0,-1),0,0);
```

```

lpVerts[3] = D3DVERTEX(D3DVECTOR(0,10,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[4] = D3DVERTEX(D3DVECTOR(3,7,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[5] = D3DVERTEX(D3DVECTOR(5,5,0),D3DVECTOR(0,0,-1),0,0);

```

It can then render the triangle fan using the **IDirect3DDevice7::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice7::DrawPrimitive** for drawing the triangle fan in the preceding example. Remember that all calls to **IDirect3DDevice7::DrawPrimitive** must occur between **IDirect3DDevice7::BeginScene** and **IDirect3DDevice7::EndScene**.

```

HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice is a valid
// pointer to an IDirect3DDevice7 interface.
hResult =
    lpDirect3DDevice->DrawPrimitive(D3DPT_TRIANGLEFAN,
                                    D3DFVF_VERTEX,
                                    lpVerts,
                                    TOTAL_VERTS,
                                    D3DDP_WAIT);

```

[\[Visual Basic\]](#)

A Visual Basic application can create a triangle fan by setting up an array of vertices as shown in the following code fragment:

```

' This code fragment assumes that dx is a valid reference
' to a DirectX7 object.
On Local Error Resume Next
Const TOTAL_VERTS As Integer = 6
Dim Verts(TOTAL_VERTS) As D3DVERTEX

```

```

Verts(0) = dx.CreateD3DVertex(0, 0, 0, 0, 0, -1, 0, 0)
Verts(1) = dx.CreateD3DVertex(-5, -5, 0, 0, 0, -1, 0, 0)
Verts(2) = dx.CreateD3DVertex(-3, 7, 0, 0, 0, -1, 0, 0)
Verts(3) = dx.CreateD3DVertex(0, 10, 0, 0, 0, -1, 0, 0)
Verts(4) = dx.CreateD3DVertex(3, 7, 0, 0, 0, -1, 0, 0)
Verts(5) = dx.CreateD3DVertex(5, 5, 0, 0, 0, -1, 0, 0)

```

It can then render the triangle fan using the **Direct3DDevice7.DrawPrimitive** method. The following code fragment illustrates the use of **Direct3DDevice7.DrawPrimitive** for drawing the triangle fan in the preceding example. Remember that all calls to **Direct3DDevice7.DrawPrimitive** must occur between **Direct3DDevice7.BeginScene** and **Direct3DDevice7.EndScene**.

```

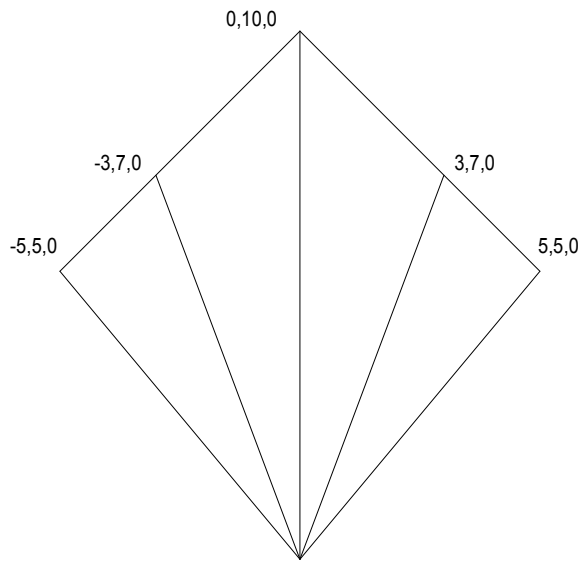
' This code fragment assumes that d3ddev is a valid
' reference to an Direct3DDevice7 object.
Call d3ddev.DrawPrimitive(D3DPT_TRIANGLEFAN, _
    D3DFVF_VERTEX, _
    Verts(0), _
    TOTAL_VERTS, _

```

```
D3DDP_WAIT)

If Err.Number <> DD_OK Then
    ' Code to handle error goes here.
End If
```

This illustration depicts the resulting triangle fan.



AGP Surfaces and Direct3D Devices

Using textures in 3-D applications can greatly enhance the appearance of realism in 3-D images. Memory for storing textures never seems as abundant as developers would like. DirectDraw and Direct3D support the Accelerated Graphics Port (AGP) architecture. The AGP architecture gives computers the ability to create DirectDraw surfaces in applications' memory spaces. This significantly extends the amount of memory available for storing textures.

Only the Direct3D HAL device supports the use of the AGP architecture. For more information on the use of the AGP architecture, see [Using Non-local Video Memory Surfaces](#).

Geometry Pipeline

When you design a 3-D application, you 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 of the objects in your world and the position and orientation of the viewer. Direct3D

transforms this description into a series of pixels on the screen. This process—the transformation of the geometry you supply into a two-dimensional image—is the geometry pipeline, sometimes called the transformation pipeline.

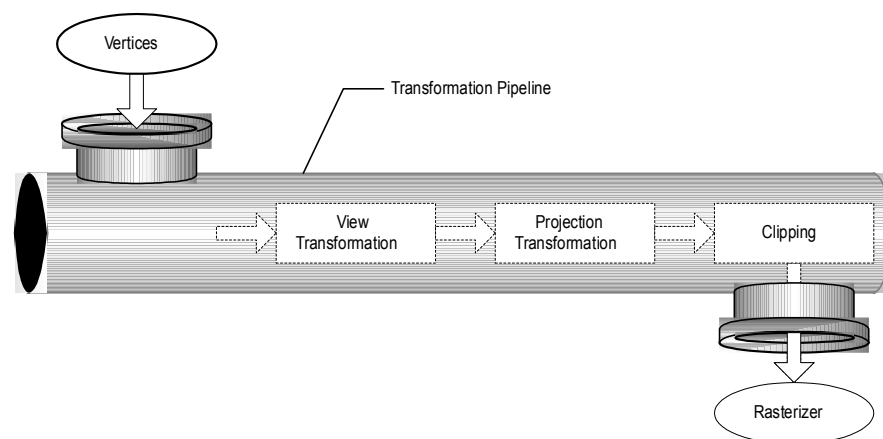
This section provides a Direct3D-centered approach to discussing the geometry pipeline. The following topics introduce key concepts and make parallels from those concepts to their counterparts in the Direct3D API:

- Overview of the Pipeline
- The World Transformation
- The View Transformation
- The Projection Transformation
- Viewports and Clipping
- The Rasterizer

Overview of the Pipeline

The part of Direct3D that pushes geometry through the 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. For more information, see *Lighting and Materials*.)

The geometry pipeline takes vertices as input. The transformation engine applies three transformations to the vertices (the world, view, and projection transformations), clips the result, and passes everything on to the rasterizer. The sequence of steps looks like this:



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. This transformation is discussed in *The World Transformation*.

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. For more information, see *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. This transformation is discussed in *The Projection Transformation*. 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, and is discussed in *Viewports and 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.

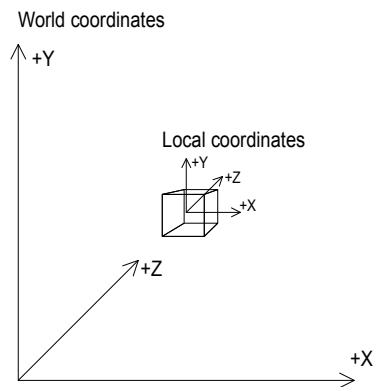
The World Transformation

The discussion of the world transformation introduces basic concepts and provides details on how to set up a world transformation matrix in a 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 of the objects in a scene. In essence, the world transformation places a model into the world; hence its name. The following illustration provides a graphical representation of 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

Like 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 in the D3dutil.cpp and D3dmath.cpp files (included with the DirectX SDK) 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
 * assumed to be valid and initialized.
 *
 * The m_vPos variable is a D3DVECTOR that contains the model's
 * location in world coordinates.
 *
 * The m_fPitch, m_fYaw, and m_fRoll variables are D3DVALUES that
 * contain the model's orientation in terms of pitch, yaw, and roll
 * angles (in radians).
 */

D3DMATRIX C3DModel::MakeWorldMatrix(void)
{
    D3DMATRIX matWorld, // World matrix being constructed.
```

```

    matTemp, // Temp matrix for rotations.
    matRot; // Final rotation matrix (applied to matWorld).

    // Using the left-to-right order of matrix concatenation,
    // apply the translation to the object's world position
    // before applying the rotations.
    D3DUtil_SetTranslateMatrix(matWorld, m_vPos);
    D3DUtil_SetIdentityMatrix(matRot);

    //
    // Now, apply the orientation variables into the
    // world matrix
    //
    if(m_fPitch || m_fYaw || m_fRoll)
    {
        // Produce and combine the rotation matrices.
        D3DUtil_SetRotateXMatrix(matTemp, m_fPitch); // pitch
        D3DMath_MatrixMultiply(matRot,matRot,matTemp);
        D3DUtil_SetRotateYMatrix(matTemp, m_fYaw); // yaw
        D3DMath_MatrixMultiply(matRot,matRot,matTemp);
        D3DUtil_SetRotateZMatrix(matTemp, m_fRoll); // roll
        D3DMath_MatrixMultiply(matRot,matRot,matTemp);

        // Apply the rotation matrices to complete the world matrix.
        D3DMath_MatrixMultiply(matWorld, matRot, matWorld);
    }
    return (matWorld);
}

```

After you prepare the world transformation matrix, call the **IDirect3DDevice7::SetTransform** method to set it, specifying the D3DTRANSFORMSTATE_WORLD flag in the first parameter. For more information, see Setting Transformations.

[\[Visual Basic\]](#)

The following example, from a fictitious 3-D model class, uses the functions offered by the **DirectX7** class 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
' assumed to be valid and initialized.
'
' The g_dx variable contains a valid reference to a DirectX7 object.
'
' The m_vPos variable is a D3DVECTOR that contains 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 be returned.
    Dim matTemp As D3DMATRIX ' Temp matrix to hold rotations.
    Dim matRot As D3DMATRIX ' Temp rotation matrix.

    ' Modify matWorld to create a translation matrix.
    Call g_dx.IdentityMatrix(matWorld)
    matWorld.rc41 = m_vPos.x
    matWorld.rc42 = m_vPos.y
    matWorld.rc43 = m_vPos.z

    Call g_dx.IdentityMatrix(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.
    Call g_dx.RotateXMatrix(matTemp, m_sPitch)
    Call g_dx.MatrixMultiply(matRot, matRot, matTemp)

    ' Then, yaw.
    Call g_dx.RotateYMatrix(matTemp, m_sYaw)
    Call g_dx.MatrixMultiply(matRot, matRot, matTemp)

    ' Finally, roll.
    Call g_dx.RotateZMatrix(matTemp, m_sRoll)
    Call g_dx.MatrixMultiply(matRot, matRot, matTemp)

    ' Apply the rotation matrices to the translation already in
    ' matWorld to complete the world matrix.
    Call g_dx.MatrixMultiply(matWorld, matRot, matWorld)
    End If

    MakeWorldMatrix = matWorld
End Function

```

After you prepare the world transformation matrix, call the **Direct3DDevice7.SetTransform** method to set it, specifying the D3DTRANSFORMSTATE_WORLD flag in the first parameter. For more information, see Setting Transformations.

Performance Optimization

Direct3D uses the world and view matrices that you set to configure several of its 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 proverbial "world-view" matrix that you set as the world matrix, then set 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 View Transformation

This section introduces the basic concepts of the view transformation and provides details on how you can set up a view transformation matrix in a 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?

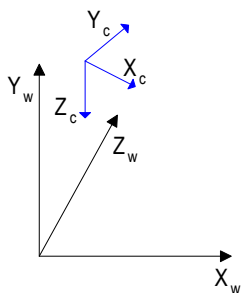
The view transformation locates the viewer in world space, transforming vertices into camera space. In camera space, the camera (or "viewer") is at the origin, looking in the positive z-direction. (Recall that Direct3D uses a left-handed coordinate system, so z is positive into a scene.) The view matrix 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 $\langle 10, 20, 100 \rangle$, the aim of the translation matrix is to move objects -10 units along the x-axis, -20 units along the y-axis, and -100 along the z-axis. The rotation matrices in the formula are based on the camera's orientation, in terms of how 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 would include 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 toward the camera, giving the appearance that the camera is above the model.

Another approach involves creating the composite view matrix directly. (The **D3DUTIL_SetViewMatrix** helper function in the D3dutil.cpp source file uses this technique, as does the **DirectX7.ViewMatrix** method used by Visual Basic applications). This approach uses the camera's world space position and a "look-at point" within 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 **D3DUtil_SetViewMatrix** helper function, from the D3dutil.cpp source file that is included with this SDK, creates a view matrix based on the camera location and a look-at point passed to it. It uses the **Magnitude**, **CrossProduct**, and **DotProduct** D3D_OVERLOADS helper functions.

```
HRESULT D3DUtil_SetViewMatrix( D3DMATRIX& mat, D3DVECTOR& vFrom,
    D3DVECTOR& vAt, D3DVECTOR& vWorldUp )
{
    // Get the z basis vector, which points straight ahead. This is the
    // difference from the eyepoint to the lookat point.
    D3DVECTOR vView = vAt - vFrom;

    FLOAT fLength = Magnitude( vView );
    if( fLength < 1e-6f )
        return E_INVALIDARG;

    // Normalize the z basis vector
    vView /= fLength;

    // Get the dot product, and calculate the projection of the z basis
    // vector onto the up vector. The projection is the y basis vector.
    FLOAT fDotProduct = DotProduct( vWorldUp, vView );

    D3DVECTOR vUp = vWorldUp - fDotProduct * vView;

    // If this vector has near-zero length because the input specified a
    // bogus up vector, let's try a default up vector
    if( 1e-6f > ( fLength = Magnitude( vUp ) ) )
    {
        vUp = D3DVECTOR( 0.0f, 1.0f, 0.0f ) - vView.y * vView;

        // If we still have near-zero length, resort to a different axis.
        if( 1e-6f > ( fLength = Magnitude( vUp ) ) )
        {
            vUp = D3DVECTOR( 0.0f, 0.0f, 1.0f ) - vView.z * vView;
```

```

    if( 1e-6f > ( fLength = Magnitude( vUp ) ) )
    return E_INVALIDARG;
}
}

// Normalize the y basis vector
vUp /= fLength;

// The x basis vector is found simply with the cross product of the y
// and z basis vectors
D3DVECTOR vRight = CrossProduct( vUp, mView );

// Start building the matrix. The first three rows contains the basis
// vectors used to rotate the view to point at the lookat point
D3DUtil_SetIdentityMatrix( mat );
mat._11 = vRight.x;  mat._12 = vUp.x;   mat._13 = mView.x;
mat._21 = vRight.y;  mat._22 = vUp.y;   mat._23 = mView.y;
mat._31 = vRight.z;  mat._32 = vUp.z;   mat._33 = mView.z;

// Do the translation values (rotations are still about the eyepoint)
mat._41 = - DotProduct( vFrom, vRight );
mat._42 = - DotProduct( vFrom, vUp );
mat._43 = - DotProduct( vFrom, mView );

return S_OK;
}

```

As with the world transformation, you call the **IDirect3DDevice7::SetTransform** method to set the view transformation, specifying the **D3DTRANSFORMSTATE_VIEW** flag in the first parameter. See [Setting Transformations](#), for more information.

[\[Visual Basic\]](#)

Visual Basic applications use the **DirectX7.ViewMatrix** method to create view matrix based on the camera location, a look-at point, an "up" vector (usually 0,1,0), and a roll value specified in radians.

The following example code, written in Visual Basic, shows how the **ViewMatrix** method can be used.

```

' For this example, the g_dx variable contains a valid reference
' to a global DirectX7 object.
Dim matView As D3DMATRIX
Dim vFrom As D3DVECTOR, _
    vTo As D3DVECTOR, _
    vUp As D3DVECTOR

```

```
vFrom.x = 0: vFrom.y = 0: vFrom.z = -15  
vTo.x = 0: vTo.y = 0: vTo.z = 0  
vUp.x = 0: vUp.y = 1: vUp.z = 0
```

' Initialize the matrix to the identity.

```
g_dx.IdentityMatrix matView
```

' Create a view matrix with a camera at 0,0,-15, that points at the

' world-origin, with no rotation around the vFrom->vTo vector.

Call `g_dx.ViewMatrix(matView, vFrom, vTo, vUp, 0)`

As with the world transformation, you call the **Direct3DDevice7.SetTransform** method to set the view transformation, specifying the `D3DTRANSFORMSTATE_VIEW` flag in the first parameter. See [Setting Transformations](#), for more information.

Performance Optimization

Direct3D uses the world and view matrices that you set to configure several of its 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, 20000 times per frame—is computationally expensive. You can minimize the number of required calculations by concatenating your world and view matrices into a proverbial "world-view" matrix that you set as the world matrix, then set 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

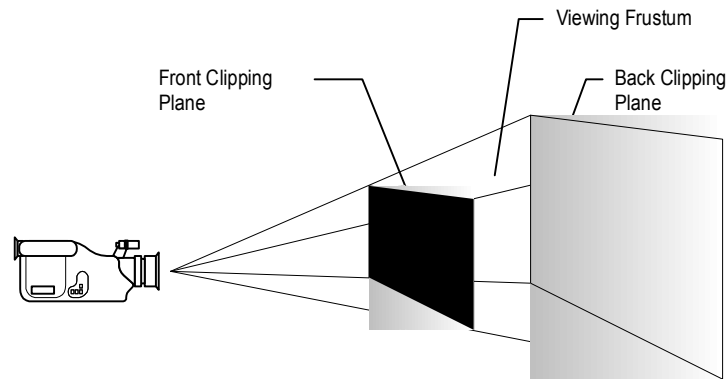
You can think of the projection transformation as controlling the camera's internals; it is analogous to choosing a lens for the camera. This is the most complicated of the three transformation types. This discussion of the projection transformation is organized into the following topics:

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

The Viewing Frustum

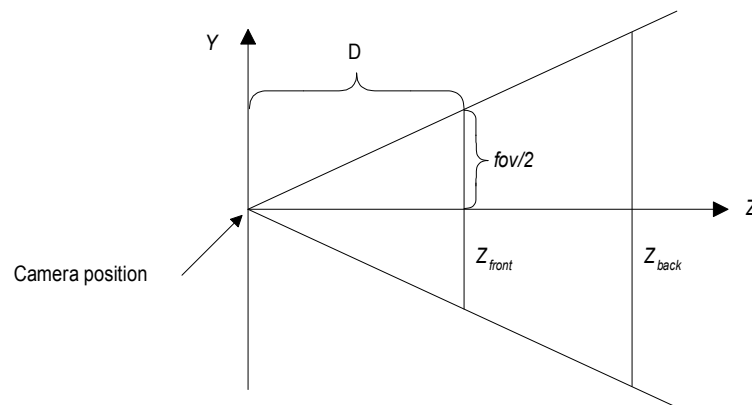
A viewing frustum is 3-D volume in a scene positioned relative to the viewport's camera. The shape of the volume affects how models are projected from camera space onto the screen. The most common type of projection, a perspective projection,

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 out 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:

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

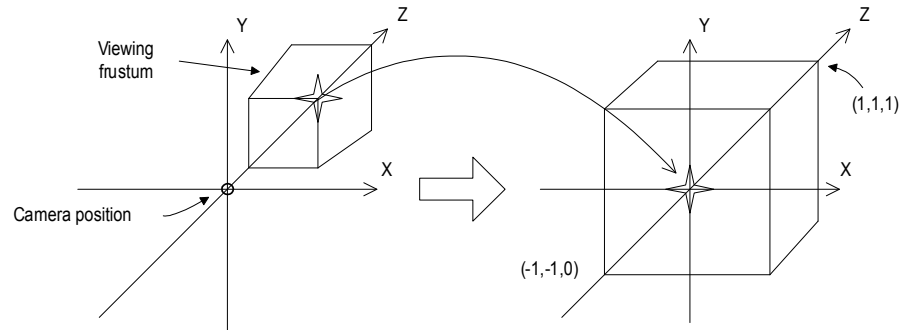
The viewing matrix puts the camera at the origin of the scene. Since the projection matrix needs to have the camera at $(0, 0, -D)$, it translates the vector by $-D$ in the z -direction, by using the following matrix:

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

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 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 (noting that fov_w and fov_h represent the viewport's horizontal and vertical fields-of-view, in radians):

$$w = \cot \left(\frac{fov_w}{2} \right)$$

$$h = \cot \left(\frac{fov_h}{2} \right)$$

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

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 **dwWidth** and **dwHeight** members of the **D3DVIEWPORT7** structure.

Whatever formula you decide to use, it's important that you set Z_n to as large a value as you can, 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 **IDirect3DDevice7::SetTransform** method to set the projection transformation; for more information, see Setting Transformations.

[Visual Basic]

For a Visual Basic application, these two dimensions correspond directly to the **IWidth** and **IHeight** members of the **D3DVIEWPORT7** type.

Whatever formula you decide to use, it's important that you set Z_n to as large a value as you can, 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 **Direct3DDevice7.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)cot(fov_horiz*0.5);
    h = (float)cot(fov_vert*0.5);
    Q = far_plane/(far_plane - near_plane);

    D3DMATRIX ret = ZeroMatrix();
    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()

```

When you have created the matrix, you need set it in a call to the **IDirect3DDevice7::SetTransform** method, specifying **D3DTRANSFORMSTATE_PROJECTION** in the first parameter. For details, see [Setting Transformations](#).

[\[Visual Basic\]](#)

Applications written in Visual Basic can create a projection matrix "by hand" as described in the [What Is the Projection Transformation?](#) topic. However, the **DirectX7** object includes an implementation of this approach, offered through the **DirectX7.ProjectionMatrix** method.

The following Visual Basic code fragment shows how this **DirectX7.ProjectionMatrix** is commonly used.

```

'
' For this example, the g_dx variable contains a reference to a global
' DirectX7 object.
'
' The pi variable is a variable of type Single that loosely approximates
' the trigonometric constant of the same name (3.141592).
Dim matProj As D3DMATRIX

' Initialize the matrix to the identity.
g_dx.IdentityMatrix matProj

' Create a projection matrix that reflects a viewing frustum with the

```

' front and rear clipping planes at 1 and 1000, and a field-of-view

' angle of pi/2 radians (90 degrees).

Call `g_dx.ProjectionMatrix(matProj, 1, 1000, pi / 2)`

When you have created the matrix, you need set it in a call to the

Direct3DDevice7.SetTransform method, specifying

`D3DTRANSFORMSTATE_PROJECTION` in the first parameter. For details, see [Setting Transformations](#).

A W-Friendly Projection Matrix

Direct3D Immediate Mode can utilize 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 will not be 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.

Viewports and Clipping

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

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

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

[C++]

The **IDirect3DDevice7** interface exposes methods that enable C++ applications to manipulate the viewport parameters for a device. (In previous releases of DirectX, viewports were implemented as a separate COM object that exposed its own interface.)

[Visual Basic]

The **Direct3DDevice7** class offers methods that enable applications written in Visual Basic to manipulate the viewport parameters for a device.

What Is a Viewport?

Conceptually, a viewport is a 2-D rectangle into which a three dimensional scene is projected. (In Direct3D, the rectangle exists as coordinates within a DirectDraw 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++]

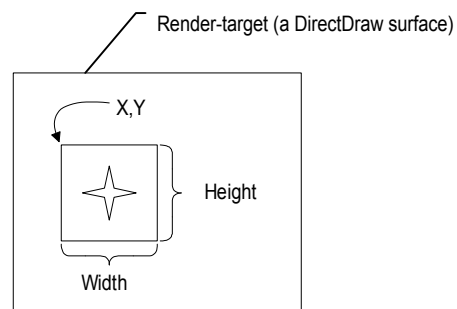
Note

The viewport structure used in DirectX 7.0 is much different than the structure recommended in previous releases (the legacy **D3DVIEWPORT2** structure).

Please read this information carefully to ensure that your application uses viewport parameters properly.

You define the viewport rectangle in C++ by using the **D3DVIEWPORT7** structure. The **D3DVIEWPORT7** structure is used with the viewport manipulation methods offered by the **IDirect3DDevice7** interface: **IDirect3DDevice7::SetViewport** and **IDirect3DDevice7::GetViewport**. The **D3DVIEWPORT7** structure contains four members—**dwX**, **dwY**, **dwWidth**, and **dwHeight**—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 **dwX**, **dwY**, **dwWidth**, and **dwHeight** members of the **D3DVIEWPORT7** structure are screen coordinates relative to the upper-left corner of the render-target surface. The structure defines two additional members (**dvMinZ** and **dvMaxZ**) indicate the depth-ranges into which the scene will be rendered.

The viewport structure used in DirectX 7.0 is much different than the structure recommended in previous releases (the legacy **D3DVIEWPORT2** structure). In short, the **D3DVIEWPORT7** structure simplifies the legacy structure by omitting the members that were used describe the size and dimensions of the viewport, in 3-D projection space. The vast majority of applications always set the now absent members to the same values, while a few used them to scale geometry to adjust for the viewport's aspect ratio. This effect was difficult to perform accurately and reliably.

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 DirectX 7.0, 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 section.

Note

The **D3DVIEWPORT7** structure members **dvMinZ** and **dvMaxZ** are interpreted in a manner completely different than in previous versions of DirectX. The **dvMinZ** and **dvMaxZ** members now 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

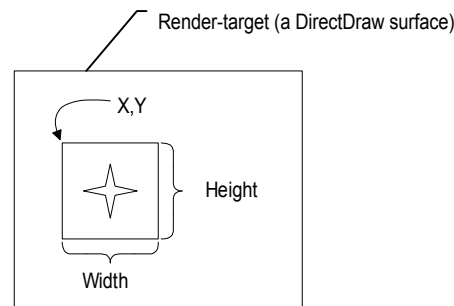
to the entire range of depth values in the depth buffer. In some cases, you can achieve special effects by using other depth ranges. For instance, if you wanted to render a heads-up display in a game, you could set both values to 0.0 to force the system to render objects in a scene in the foreground,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

[Visual Basic]

Within a Visual Basic application, you define the viewport rectangle by using the **D3DVIEWPORT7** type. The **D3DVIEWPORT7** type is used with the viewport manipulation methods offered by the **Direct3DDevice7** class:

Direct3DDevice7.SetViewport and **Direct3DDevice7.GetViewport**. The **D3DVIEWPORT7** type contains four members—**IX**, **IY**, **IWidth**, and **IHeight**—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 **IX**, **IY**, **IWidth**, and **IHeight** members of the **D3DVIEWPORT7** 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.

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 DirectX 7.0, 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 The Projection Transformation section.

Clipping Volumes

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

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

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 `D3DRENDERSTATE_CLIPPING` render state. Clipping information for vertex buffers is generated during processing, for more information see Processing Vertices.

[Visual Basic]

With the exception of vertex buffers, applications enable or disable clipping by way of the `D3DRENDERSTATE_CLIPPING` render state. Clipping information for vertex buffers is generated during processing, for more information see Processing Vertices.

Viewport Scaling

[C++]

The dimensions used in the **dwX**, **dwY**, **dwWidth**, and **dwHeight** members of the **D3DVIEWPORT7** 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.

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 The Rasterizer.

Note

The viewport scaling matrix in by DirectX 7.0 incorporates the **dvMinZ** and **dvMaxZ** members of the **D3DVIEWPORT7** structure to scale vertices to fit the depth range [**dvMinZ**, **dvMaxZ**]. This represents different semantics from previous releases of DirectX, in which these members were used for clipping. For more information, see Viewport Rectangle and Clipping Volumes. Applications typically set **dvMinZ** and **dvMaxZ** 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 **lX**, **lY**, **lWidth**, and **lHeight** members of the **D3DVIEWPORT7** 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.

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 The Rasterizer.

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 thing required to configure 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 could use the following settings for the members of the **D3DVIEWPORT7** structure to achieve this from C++:

```
memset(&viewData, 0, sizeof(D3DVIEWPORT7));
viewData.dwX = 0;
viewData.dwY = 0;
viewData.dwWidth = width;
viewData.dwHeight = height;
viewData.dvMinZ = 0.0f;
viewData.dvMaxZ = 1.0f;
```

After setting values in the **D3DVIEWPORT7** structure, simply apply the viewport parameters to the device by calling its **IDirect3DDevice7::SetViewport** method. The following examples shows what this call might look like:

```
HRESULT hr;

hr = lpD3DDevice->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. If you need to make changes to the viewport parameters, just update the values in the **D3DVIEWPORT7** structure and call **SetViewport** again.

Note

The **D3DVIEWPORT7** structure members **dvMinZ** and **dvMaxZ** are interpreted in a manner completely different than in previous versions of DirectX. The **dvMinZ** and **dvMaxZ** members now 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, if you wanted to render a heads-up display in a game, you could 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 Visual Basic code creates settings in a **D3DVIEWPORT7** type to achieve this:

```
Dim viewData As D3DVIEWPORT7
```

```
With viewData
```

```
    .IX = 0: .IY = 0
```

```
    .IWidth = Width: .IHeight = Height
```

```
    .minz = 0#: .maxz = 1#
```

```
End With
```

After setting values in the **D3DVIEWPORT7** type, simply apply the viewport parameters to the device by calling its **Direct3DDevice7.SetViewport** method.

' The dev variable contains a valid reference to a Direct3DDevice7 object.

```
Call dev.SetViewport(viewData)
```

After the call, the viewport parameters are set and will take effect the next time a rendering method is called. If you need to make changes to the viewport parameters, just update the values in the **D3DVIEWPORT7** 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 will 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 **IDirect3DDevice7** interface offers C++ developers the **IDirect3DDevice7::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, perhaps—you might want to clear the entire viewport each frame. In this situation, you would set the *dwCount* parameter to 1, and the *lpRects* parameter to the address of a single rectangle that covers the entire viewport area. If you find it more convenient, you can set the *lpRects* parameter to NULL and the *dwCount* parameter to 0 to indicate that the entire viewport rectangle should be cleared.

The **Clear** method is flexible, and provides support for clearing stencil bits within a depth buffer. The *dwFlags* 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 *dwColor* 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 *dvZ*: 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 *dwStencil* parameter. You can use integers that range from 0 to 2^n-1 , where n is the stencil buffer bit depth.

Note

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. So long as your application clears the depth buffer, all pixels on the render-target surface will be overwritten anyway.

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

[\[Visual Basic\]](#)

The **Direct3DDevice7** Visual Basic class offers the **Direct3DDevice7.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, perhaps—you might want to clear the entire viewport each frame. In this situation, you would set the *count* parameter to 1, and the *recs()* 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 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.

In some situations, you might only be rendering to small portions of the render target and depth buffer surfaces. The clear methods also allow 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 *recs()* parameter.

Manually Transforming Vertices

You can use three different kinds of vertices in your Direct3D application. Read [Vertex Formats](#) for more details on the vertex formats:

[\[C++\]](#)

Untransformed, unlit vertices

Vertices that your application doesn't light or transform. C++ applications that neither transform nor light vertices before rendering a scene can use **D3DVERTEX** vertices (or an equivalent flexible vertex format). Although you specify lighting parameters and transformation matrices, Direct3D does the math.

Untransformed, lit vertices

Vertices that your application lights but does not transform. C++ applications that use customized lighting effects might use **D3DLVERTEX** vertices (or an equivalent flexible vertex format).

Transformed, lit, vertices

Vertices that your application both lights and transforms. C++ applications that transform and light vertices on their own might use **D3DTLVERTEX** vertices (or an equivalent flexible vertex format). These vertices skip the transformations in the geometry pipeline altogether but they can be clipped by the system if needed. If your application clips vertices itself, you can get the best performance by specifying the **D3DDP_DONOTCLIP** flag when calling a rendering method. If you want Direct3D to clip vertices, omit the **D3DDP_DONOTCLIP** flag. Note that if you request Direct3D clipping on transformed and lit vertices, the system back-transforms them to camera space for clipping, then transforms them back to screen space, incurring processing overhead.

In DirectX 7.0, you can change from simple to complex vertex types by using vertex buffers. (In previous releases of DirectX, applications could call the **TransformVertices** method of the legacy **IDirect3DViewport3** interface. With the advent of vertex buffers in DirectX 6.0 and the abandonment of discrete viewport objects in DirectX 7.0, this method became obsolete.) Vertex buffers are objects used to efficiently contain and process batches of vertices for rapid rendering, and are optimized to exploit processor-specific features. Vertex buffers offer the **IDirect3DVertexBuffer7::ProcessVertices** and **IDirect3DVertexBuffer7::ProcessVerticesStrided** methods to perform vertex transformations for you; this is usually much faster than the **TransformVertices** method. The **ProcessVertices** family of methods method 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 actually 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, unlit vertices

Vertices that your application doesn't light or transform. Visual Basic applications that neither transform nor light vertices before rendering a scene can use **D3DVERTEX** vertices (or an equivalent flexible vertex format). Although you specify lighting parameters and transformation matrices, Direct3D does the math.

Untransformed, lit vertices

Vertices that your application lights but does not transform. Visual Basic applications that use customized lighting effects might use **D3DLVERTEX** vertices (or an equivalent flexible vertex format).

Transformed, lit, vertices

Vertices that your application both lights and transforms. Visual Basic applications that transform and light vertices on their own might use **D3DTLVERTEX** vertices (or an equivalent flexible vertex format). These vertices skip the transformations in the geometry pipeline altogether but they can be clipped by the system if needed. If your application clips vertices itself, you can get the best performance by specifying the **D3DDP_DONOTCLIP** flag when calling a rendering method. If you want Direct3D to clip vertices, omit the **D3DDP_DONOTCLIP** flag. Note that if you request Direct3D clipping on transformed and lit vertices, the system back-transforms them to camera space for clipping, then transforms them back to screen space, incurring processing overhead.

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. Vertex buffers offer the **Direct3DVertexBuffer7.ProcessVertices** method to perform vertex transformations for you; this is usually much faster than the **TransformVertices** method. The **ProcessVertices** method accepts only untransformed vertices, and can optionally light and clip vertices as well. Lighting is performed at the time you call **ProcessVertices**, but clipping is actually 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](#).

The Rasterizer

After passing through the 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 be sent to the rasterizer to be painted 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,

and 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 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).

Lighting and Materials

This section describes illumination in Direct3D Immediate Mode scenes, using ambient light levels, and applying lights and materials to objects in your 3-D scenes. The following topics are discussed:

- **Introduction to Lighting and Materials**
Describes, in general terms, the roles of light sources and materials in a Direct3D Immediate Mode scene.
- **Direct3D Light Model vs. Nature**
Describes the Direct3D illumination model in more detail, and contrasts it with the behavior of light in the natural world.
- **Color Values for Lights and Materials**
Describes the RGBA color values used with both light sources and materials in Direct3D Immediate Mode.
- **Direct Light vs. Ambient Light**
Compares and contrasts the roles of true light sources and ambient light levels.
- **Enabling and Disabling the Lighting Engine**
Describes how to enable or disable the Direct3D lighting engine.
- **Lights**
Describes lights in Direct3D and provides information on how to use them in a scene.
- **Materials**
Describes materials in Direct3D Immediate Mode and provides information about how to use them in a scene.
- **Mathematics of Direct3D Lighting**

Provides details about the math behind the Direct3D light model.

Introduction to Lighting and Materials

When lighting is enabled, as 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, as well as 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 are allowing Direct3D to handle the details of illumination for you, but advanced users can perform lighting on their own if necessary.

How you work with lighting and materials makes a huge difference in the appearance of the rendered scene. Materials define how light reflects off of a surface. Direct light and ambient light levels define the light that is being reflected. You must use materials to render a scene if you are letting Direct3D handle lighting. Lights are not actually required to render a scene, but you'll be hard pressed to see much in a scene rendered without light. At best, rendering an unlit scene will result in a silhouette of the objects in the scene—not enough detail for most purposes.

Direct3D Light Model vs. Nature

In nature, when light is emitted from a source, it reflects off hundreds (if not thousands or millions) of objects before reaching the viewer's eye. Each time it reflects, parts of the light are absorbed by a surface, parts are scattered in random directions, and the rest goes on to another surface or to the viewer. This process continues until the light attenuates to nothing or a viewer perceives the light—but, who knows how many times the light will bounce? It could be once, a hundred times, or millions of times.

Obviously, the calculations required to perfectly simulate the natural behavior of light are, by far, too time consuming to be used for real-time 3-D graphics. Therefore, with the interest of speed in mind, the 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 drawn to the screen. For specific information about the algorithms Direct3D uses, see *Mathematics of Direct3D Lighting*.

The Direct3D Immediate Mode 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 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

Direct3D Immediate Mode describes color in terms the 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. (Visual Basic applications use the **D3DCOLORVALUE** type.) 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 within the structure are used a little differently by each.

Color values for light sources represent the amount of a particular light component it emits. Lights don't use an alpha component, so you only need to think about the red, green, and blue components of the color. 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 some level in between. The colors coming from each lens 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 results in a light that doesn't emit light at all. You can make a light that emits only one component, resulting in a purely red, green, or blue light, or, the light could use combinations to emit colors like yellow or purple. You can even set negative values 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 given 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 will reflect all the light that comes its way. Likewise, a material with R: 0.0, G: 1.0, B: 0.0, A: 1.0 will reflect all of 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 than 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 sets of light parameters for direct light by calling the **IDirect3DDevice7::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 within 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 **IDirect3DDevice7::SetRenderState** method, specifying **D3DRENDERSTATE_AMBIENT** as the *dwRenderStateType* parameter, and the desired RGBA color as the *dwRenderState* 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 Direct3D Immediate Mode. For more information, see [Color Values for Lights and Materials](#).) You can use the **RGBA_MAKE** macro to generate RGBA values from integers. 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. In ramp emulation, ambient light doesn't have color, so the alpha component is used for brightness. 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 sets of light parameters for direct light by calling the **Direct3DDevice7.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 within 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 **Direct3DDevice7.SetRenderState** method, specifying **D3DRENDERSTATE_AMBIENT** as the *state* parameter, and the desired RGBA color as the *renderstate* parameter.

You can use the **DirectX7.CreateColorRGBA** method to generate RGBA values from values of type **Single**. 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. In ramp emulation, ambient light doesn't have color, so the alpha component is used for brightness. When using hardware acceleration or RGB emulation, the alpha component is ignored.

Enabling and Disabling the Lighting Engine

[C++]

By default, Direct3D performs lighting calculations on all vertices, even those without vertex normals. (This is different from the behavior in previous releases of DirectX, where lighting was performed only on vertices that contained a vertex normal.) However, you can disable lighting through the `D3DRENDERSTATE_LIGHTING` render state. Call the **IDirect3DDevice7::SetRenderState** method, passing `D3DRENDERSTATE_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.

If your application uses vertex buffers, the process is slightly different, but it's just as simple: include or omit the `D3DVOP_LIGHT` flag when calling the **IDirect3DVertexBuffer7::ProcessVertices** or **IDirect3DVertexBuffer7::ProcessVerticesStrided** methods to enable or disable lighting for that vertex buffer.

[Visual Basic]

By default, Direct3D performs lighting calculations on all vertices, even those without vertex normals. However, you can disable lighting through the `D3DRENDERSTATE_LIGHTING` render state. Call the **Direct3DDevice7.SetRenderState** method, passing `D3DRENDERSTATE_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.

If your application uses vertex buffers, the process is slightly different, but it's just as simple: include or omit the `D3DVOP_LIGHT` flag when calling the **Direct3DVertexBuffer7.ProcessVertices** method to enable or disable lighting for that vertex buffer.

Lights

Lights are used to illuminate objects in a scene. This section describes lights and how they are used in Direct3D Immediate Mode. The following topics are discussed:

- Introduction to Light Objects

- Light Properties
- Using Lights

Introduction to Light Objects

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

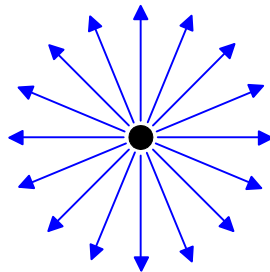
Note

DirectX 7.0 does not use the parallel-point light type offered in previous releases of DirectX.

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

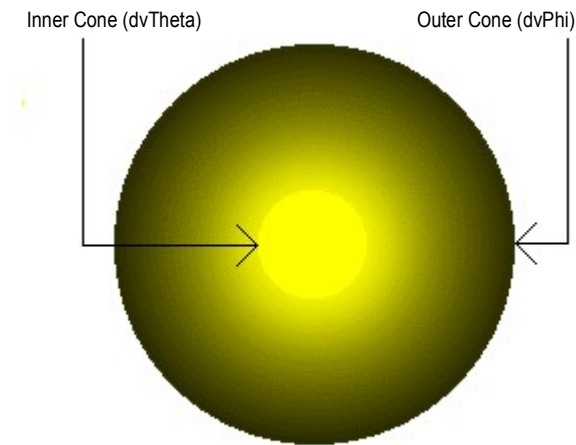
Point lights have color and position within a scene, but no single direction. They give off light equally in all directions, as shown in the following illustration.



A light bulb would be 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, 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 of these 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, along with the related members from the **D3DLIGHT7** C++ structure (the **D3DLIGHT7** Visual Basic type parallels this structure).



Spotlights are affected by falloff, attenuation, and range. These factors, as well as the distance light travels to each vertex, are figured in while computing lighting effects for objects in a scene. Computing all these effects for each vertex makes spotlights the most computationally expensive of all lights in Direct3D Immediate Mode.

Directional Lights

Directional lights have only color and direction, not position. They give off parallel light, meaning that all light generated by it travels through scene in the same direction. You can 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 Direct3D calculates vertex colors. Because of the small number of illumination factors, these are the least computationally intensive lights to use.

Light Properties

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 will use all properties. Direct3D Immediate Mode uses the **D3DLIGHT7** structure (for C++) and the **D3DLIGHT7** type (for 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
- 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 **dltType** member of the light's **D3DLIGHT7** structure. There are three types of lights in Direct3D Immediate Mode—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 [Introduction to 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** Visual Basic enumeration in the **dltType** member of the light's **D3DLIGHT7** type. There are three types of lights in Direct3D Immediate Mode—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 [Introduction to Light Objects](#).

Light Color

Lights in 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 **D3DLIGHT7** structure includes three members for these colors—**dcvDiffuse**, **dcvAmbient**, and **dcvSpecular**—each one is a **D3DCOLORVALUE** structure that defines the color being emitted.

[Visual Basic]

The **D3DLIGHT7** Visual Basic type includes three members for these colors—**dcvDiffuse**, **dcvAmbient**, and **dcvSpecular**—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 was "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.

Light Position, Range and Attenuation

[C++]

The position, range, and attenuation properties are used to define a light's location in world space, and how the light it emits behaves over distance. Like all light the properties you use in C++, these are carried within a light's **D3DLIGHT7** structure.

Position

Light position is described using a **D3DVECTOR** structure in the **dvPosition** member of the **D3DLIGHT7** 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 **dvRange** member contains a floating-point value that represents the light's maximum range, in world space. Most applications set the range to the maximum possible value, **D3DLIGHT_RANGE_MAX**, which is defined in D3d.h. 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. Light attenuation is represented by three **D3DLIGHT7** structure members: **dvAttenuation0**, **dvAttenuation1**, and **dvAttenuation2**. These members contain floating point values typically ranging from 0.0 to 1.0, controlling a light's constant, linear, and quadratic attenuation. Many applications set the **dvAttenuation1** member to 1.0 and the others to 0.0, resulting in light intensity that attenuates evenly over distance—from maximum intensity at the source, to zero intensity at the light's range. You can combine attenuation values to get more complex attenuation effects. Or, you might set them to values outside of 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 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 **D3DLIGHT7** type includes members that your 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 **D3DLIGHT7** 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. Most applications set the range to the maximum possible value (18.446+e18). 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. Light attenuation is represented by three **D3DLIGHT7** members: **attenuation0**, **attenuation1**, and **attenuation2**. These members contain floating point values typically ranging from 0.0 to 1.0, controlling a light's constant, linear, and quadratic attenuation. Many applications set the **attenuation1** member to 1.0 and the others to 0.0, resulting in light intensity that attenuates evenly over distance—from maximum intensity at the source, to zero intensity at the light's range. You can combine attenuation values to get more complex attenuation effects. Or, you might set them to values outside of 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 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 only used by directional and spotlights, and is described with a vector.

[C++]

An application that uses C++ employs the the **dvDirection** member (a **D3DVECTOR** structure) of the **D3DLIGHT7** structure. 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) would have a direction vector of $\langle 0, 0, 1 \rangle$ no matter where its position is defined to be. Similarly, you could simulate sunlight shining directly down on a scene by using a directional light whose direction is $\langle 0, -1, 0 \rangle$. 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]

Visual Basic applications set the light direction in the **direction** member of the light's **D3DLIGHT7** 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) would have a direction vector of $\langle 0, 0, 1 \rangle$ no matter where its position is defined to be. Similarly, you could simulate sunlight shining directly down on a scene by using a directional light whose direction is $\langle 0, -1, 0 \rangle$. 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 $\langle 0, 0, 0 \rangle$ direction vector.

Spotlight Properties

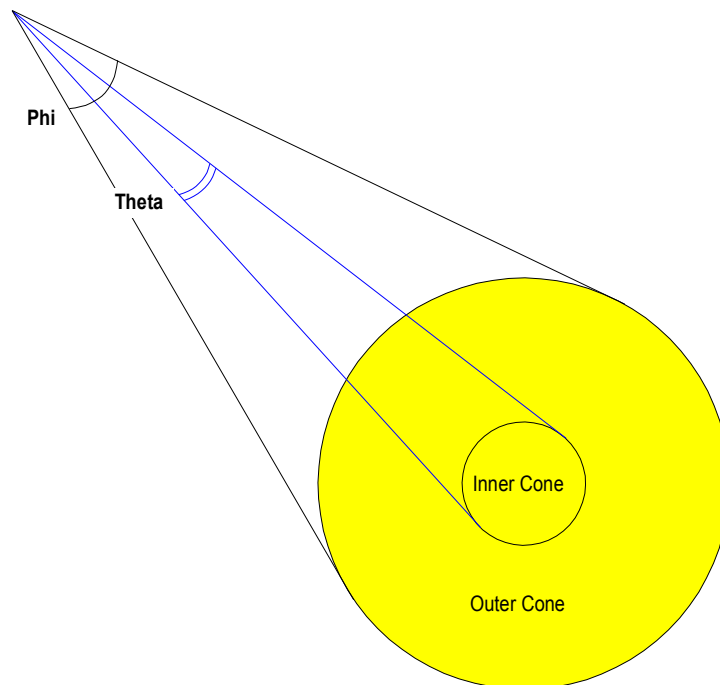
[C++]

The **D3DLIGHT7** C++ structure contains three members that are used only by spotlights. These members (**dvFalloff**, **dvTheta**, and **dvPhi**) 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 **dvTheta** value is the radian angle of the spotlight's inner cone and the **dvPhi** value is the angle for the outer cone of light. The **dvFalloff** value controls how light intensity decreases between the outer edge of the inner cone and in the inner edge of

the outer cone. Most applications will set **dvFalloff** 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 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.

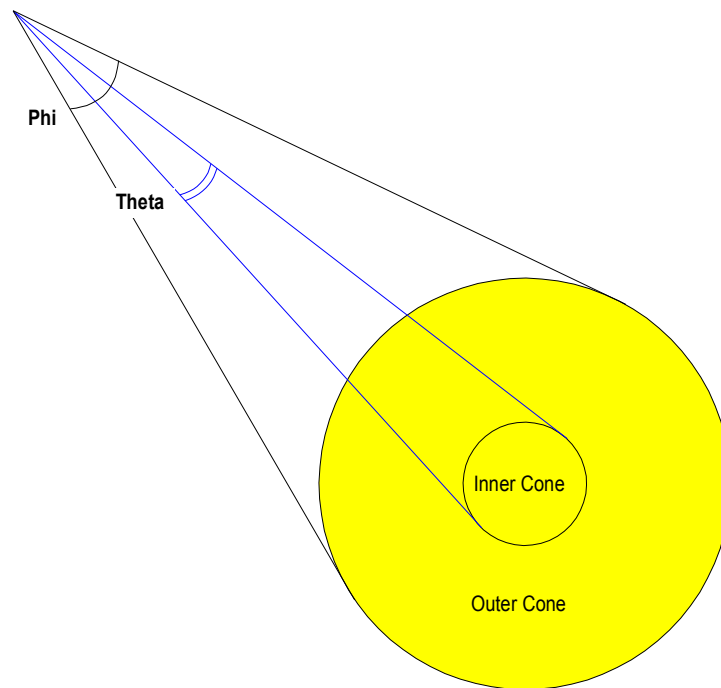


[Visual Basic]

The **D3DLIGHT7** 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 in the inner edge of the outer cone. Most applications will 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 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 Direct3D Immediate Mode 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 **D3DLIGHT7** structure and then calling the **IDirect3DDevice7::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 **D3DLIGHT7** 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 very large number of lights, with only a subset of the assigned lights enabled at a time. Check the **dwMaxActiveLights** member of the **D3DDEVICEDESC7** structure when you retrieve device capabilities to determine

the maximum number of active lights supported by that device. If you do not need to use a particular light anymore, you can disable it, or overwrite it with a new set of light properties.

The following C++ code prepares and sets properties for a white point-light whose emitted light will not attenuate over distance:

```
/*
 * For the purposes of this example, the g_lpd3dDev variable
 * is a valid pointer to an IDirect3DDevice7 interface.
 */
D3DLIGHT7 d3dLight;
HRESULT hr;

// Initialize the structure.
ZeroMemory(&d3dLight, sizeof(D3DLIGHT7));

// Set up for a white point light.
d3dLight.dltType = D3DLIGHT_POINT;
d3dLight.dcvDiffuse.r = 1.0f;
d3dLight.dcvDiffuse.g = 1.0f;
d3dLight.dcvDiffuse.b = 1.0f;
d3dLight.dcvAmbient.r = 1.0f;
d3dLight.dcvAmbient.g = 1.0f;
d3dLight.dcvAmbient.b = 1.0f;
d3dLight.dcvSpecular.r = 1.0f;
d3dLight.dcvSpecular.g = 1.0f;
d3dLight.dcvSpecular.b = 1.0f;

// Position it high in the scene, and behind the viewer.
// (Remember, these coordinates are in world space, so
// the "viewer" could be anywhere in world space, too.
// For the purposes of this example, assume the viewer
// is at the origin of world space.)
d3dLight.dvPosition.x = 0.0f;
d3dLight.dvPosition.y = 1000.0f;
d3dLight.dvPosition.z = -100.0f;

// Don't attenuate.
d3dLight.dvAttenuation0 = 1.0f;
d3dLight.dvRange = D3DLIGHT_RANGE_MAX;

// Set the property info for the first light.
hr = g_lpd3dDev->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 you want to update and the address of the **D3DLIGHT7** 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 **IDirect3DDevice7::LightEnable** method for the device.

[Visual Basic]

Your Visual Basic application sets lighting properties by preparing a **D3DLIGHT7** type and then calling the **Direct3DDevice7.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 **D3DLIGHT7** 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 been assigned properties. Applications can set a very large number of lights, with only a subset of the assigned lights enabled at a time. Check the **IMaxActiveLights** member of the **D3DDEVICEDESC7** type when you retrieve device capabilities to determine the maximum number of active lights supported by that device. If you do not need to use a particular light anymore, you can disable it, or overwrite it with a new set of light properties.

The following code, written in Visual Basic, 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 Direct3DDevice7 object.
'

Dim LightDesc As D3DLIGHT7
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
    .dltType = 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. Just specify the index of the set of light properties you want to update and the address of the **D3DLIGHT7** 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 **Direct3DDevice7.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 **IDirect3DDevice7::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 fragment 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 g_lpd3dDev variable
 * is a valid pointer to an IDirect3DDevice7 interface.
 */
HRESULT hr;

hr = g_lpd3dDev->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 **Direct3DDevice7.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 fragment 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 Direct3DDevice7 object.
'

On Local Error Resume Next

Call d3dDevice.LightEnable(0, True)
If Err.Number <> DD_OK Then
    'Code to handle the error goes here.
End If
```

If you enable a light that has no properties, or supply an index outside the range of the light property sets assigned within the device, the **LightEnable** method creates a light source with the following properties and enables or disables it:

Member	Default
dltType	D3DLIGHT_DIRECTIONAL
dcvDiffuse	(R:1, G:1, B:1, A:0)
dcvSpecular	(R:0, G:0, B:0, A:0)
dcvAmbient	(R:0, G:0, B:0, A:0)
dvPosition	(0, 0, 0)
dvDirection	(0, 0, 1)
dvRange	0
dvFalloff	0
dvAttenuation0	0
dvAttenuation1	0
dvAttenuation2	0
dvTheta	0
dvPhi	0

Retrieving Light Properties

[C++]

You can retrieve all the properties for an existing light source from C++ by calling the **IDirect3DDevice7::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 **D3DLIGHT7** structure in the second parameter. The device will fill the **D3DLIGHT7** structure to describe the lighting properties it uses for the light source at that index.

The following C++ code illustrates this process:

```
/*
 * For the purposes of this example, the g_lpd3dDev variable
 * is a valid pointer to an IDirect3DDevice7 interface.
 */
HRESULT hr;
D3DLIGHT7 light;

// Get the property info for the first light.
hr = g_lpd3dDev->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 within the device, the **GetLight** method will fail, returning **DDERR_INVALIDPARAMS**.

[\[Visual Basic\]](#)

A Visual Basic application retrieves the properties for an existing light source by calling the **Direct3DDevice7.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 **D3DLIGHT7** as the second parameter. The device will fill the **D3DLIGHT7** type to describe the lighting properties it uses for the light source at that index.

The following Visual Basic code fragment illustrates this process:

```
'
' For the purposes of this example, the d3dDevice variable contains
' a valid reference to a Direct3DDevice7 object.
'

Dim lightDesc As D3DLIGHT7

' Get the property info for the first light.
Call d3dDevice.GetLight(0, lightDesc)
If Err.Number <> DD_OK Then
    ' Code to handle the error goes here.
End If
```

If you supply an index outside the range of the light sources assigned within the device, the **GetLight** method will fail, and the value of **Err.Number** will be **DDERR_INVALIDPARAMS**.

Materials

This section describes materials and how they are used in Direct3D Immediate Mode applications. The following topics are discussed:

- What Are Materials?
- Material Properties
- Using Materials

Default Material Properties

If your application does not specify material properties for rendering, the system uses a default material. The default material reflects all diffuse light (white, you might say), 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 Direct3D the following things about the polygons it is rendering:

- How they reflect ambient and diffuse light
- What their specular highlights look like
- Whether or not the polygons appear to emit light

[C++]

Direct3D Immediate Mode applications written in C++ use the **D3DMATERIAL7** structure to describe material properties. For more information, see Material Properties.

[Visual Basic]

Direct3D Immediate Mode applications written in Visual Basic use the **D3DMATERIAL7** type to describe material properties. For more information, see Material Properties.

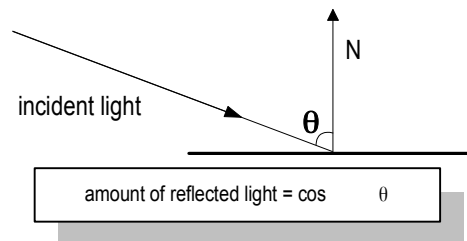
Material Properties

[C++]

Material properties detail a material's diffuse reflection, ambient reflection, light emission, and specular highlighting characteristics. Direct3D uses the **D3DMATERIAL7** structure 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 of the properties 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 **dcvDiffuse** and **dcvAmbient** members of the **D3DMATERIAL7** 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 **IDirect3DDevice7::SetRenderState** method with the **D3DRENDERSTATE_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 would create a material that reflected 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

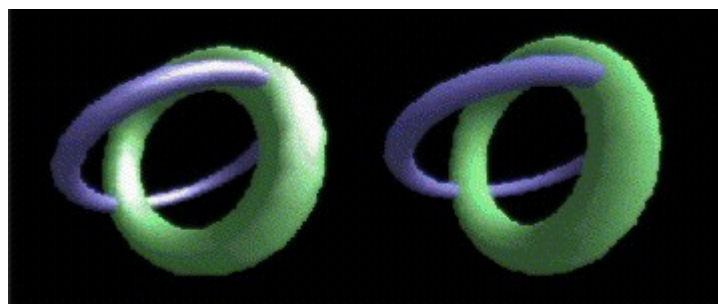
Materials can be used to make a rendered object appear to be self-luminous. The **dcvEmissive** member of the **D3DMATERIAL7** structure is used to describe the color and transparency of the emitted light. Emission affects an object's color and can, for example, make a dark material brighter and take 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 could be handy if you wanted to make the crystal appear to light up, but not actually cast light on other objects in the scene. Remember, materials with emissive properties don't actually emit light that can be reflected by other objects in a scene. To achieve this effect, you would need to place an additional light within the scene.

Specular Reflection

Specular reflection creates highlights on objects, making them appear shiny. The **D3DMATERIAL7** 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 **dcvSpecular** member to the desired RGBA color—the most common colors are white or light gray. The values you set in the **dvPower** member control how sharp the specular effects are.

Specular highlights can create dramatic effects. Drawing again on the blue crystal analogy: a larger **dvPower** value will create sharper specular highlights, making the crystal appear to be quite shiny. Smaller values increase the area of the effect, creating a dull reflection that might make the crystal look frosty. To make an object truly matte, set the **dvPower** member to zero, and the color in **dcvSpecular** 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.

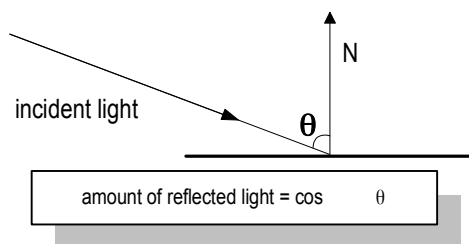


[Visual Basic]

Material properties detail a material's diffuse reflection, ambient reflection, light emission, and specular highlighting characteristics. Direct3D uses the **D3DMATERIAL7** 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 of the properties 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 **D3DMATERIAL7** 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 **Direct3DDevice7.SetRenderState** method with the **D3DRENDERSTATE_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 would create a material that reflected 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

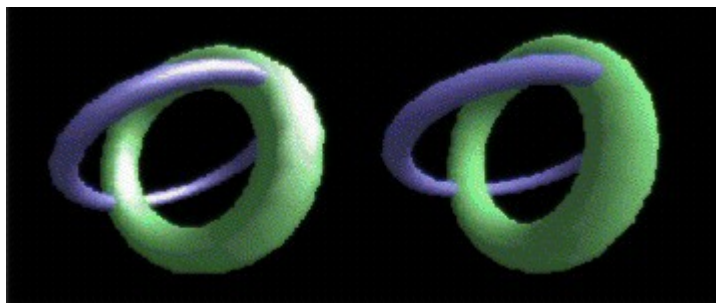
Materials can be used to make a rendered object appear to be self-luminous. The **emissive** member of the **D3DMATERIAL7** type is used to describe the color and transparency of the emitted light. Emission affects an object's color and can, for example, make a dark material brighter and take on part 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 could be handy if you wanted to make the crystal appear to light up, but not actually cast light on other objects in the scene. Remember, materials with emissive properties don't actually emit light that can be reflected by other objects in a scene. To achieve this effect, you would need to place an additional light within the scene.

Specular Reflection

Specular reflection creates highlights on objects, making them appear shiny. The **D3DMATERIAL7** 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 will create sharper specular highlights, making the crystal appear to be quite shiny. Smaller values increase the area of the effect, creating a dull reflection that might 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 Direct3D application. Information is divided into the following topics:

- Setting Material Properties
- Retrieving Material Properties

Setting Material Properties

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 you want the system to use by preparing a **D3DMATERIAL7** structure, then calling the **IDirect3DDevice7::SetMaterial** method.

To prepare the **D3DMATERIAL7** structure for use, set the property information in the structure to create the desired effect during rendering. The following code

fragment sets up the **D3DMATERIAL7** structure for a purple material with sharp white specular highlights:

```
D3DMATERIAL7 mat;

// Set the RGBA for diffuse reflection.
mat.dcvDiffuse.r = (D3DVALUE)0.5;
mat.dcvDiffuse.g = (D3DVALUE)0.0;
mat.dcvDiffuse.b = (D3DVALUE)0.5;
mat.dcvDiffuse.a = (D3DVALUE)1.0;

// Set the RGBA for ambient reflection.
mat.dcvAmbient.r = (D3DVALUE)0.5;
mat.dcvAmbient.g = (D3DVALUE)0.0;
mat.dcvAmbient.b = (D3DVALUE)0.5;
mat.dcvAmbient.a = (D3DVALUE)1.0;

// Set the color and sharpness of specular highlights.
mat.dcvSpecular.r = (D3DVALUE)1.0;
mat.dcvSpecular.g = (D3DVALUE)1.0;
mat.dcvSpecular.b = (D3DVALUE)1.0;
mat.dcvSpecular.a = (D3DVALUE)1.0;
mat.dvPower = (float)50.0;
```

After preparing the **D3DMATERIAL7** structure, you apply them by calling the **IDirect3DDevice7::SetMaterial** method of the rendering device. This method accepts the address of a prepared **D3DMATERIAL7** structure as its only parameter. You can call **SetMaterial** with new information as needed to update the material properties for the device. The following example shows how this might look in code:

```
// This code fragment uses the material properties defined for
// the mat variable earlier in this topic. The lpd3dDev is assumed
// to be a valid pointer to an IDirect3DDevice7 interface.
HRESULT hr;
hr = lpd3dDev->SetMaterial(&mat);
if(FAILED(hr))
{
    // Code to handle the error goes here.
}
```

[\[Visual Basic\]](#)

From Visual Basic, set the material properties that you want the system to use by preparing a **D3DMATERIAL7** type, then calling the **Direct3DDevice7.SetMaterial** method.

To prepare the **D3DMATERIAL7** type for use, set the property information in the structure to create the desired effect during rendering. The following code fragment

sets up the **D3DMATERIAL7** type for a purple material with sharp white specular highlights:

```
Dim mat As D3DMATERIAL7

' 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 **D3DMATERIAL7** type, you apply the properties by calling the **Direct3DDevice7.SetMaterial** method of the rendering device. This method accepts the address of a prepared **D3DMATERIAL7** type as its only parameter. You can call **SetMaterial** with new information as needed to update the material properties for the device. The following example shows how this might look in code:

```
' This code fragment uses the material properties defined for
' the mat variable earlier in this topic. The d3dDev is assumed
' to contain a valid reference to a Direct3DDevice7 object.
On Local Error Resume Next

Call d3dDev.SetMaterial(mat)
If Err.Number <> DD_OK Then
' Code to handle the error goes here.
End If
```

Retrieving Material Properties

[C++]

You retrieve the material properties that the rendering device is currently using by calling the **IDirect3DDevice7::GetMaterial** method for the device. Unlike the

IDirect3DDevice7::SetMaterial method, **GetMaterial** doesn't require preparation. The **GetMaterial** method accepts the address of a **D3DMATERIAL7** structure, and fills the provided structure with information describing the current material properties before returning.

```
// For this example, the lpd3dDev variable is assumed to
// be a valid pointer to an IDirect3DDevice7 interface.
HRESULT hr;
D3DMATERIAL7 mat;

hr = lpd3dDev->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 **Direct3DDevice7.GetMaterial** method for the device. Unlike the **Direct3DDevice7.SetMaterial** method, **GetMaterial** doesn't require preparation. The **GetMaterial** method accepts a variable of type **D3DMATERIAL7**, and fills it with information describing the current material properties before returning.

```
' For this example, the d3dDev variable is assumed to
' contain a valid reference to a Direct3DDevice7 object.
On Local Error Resume Next
Dim mat As D3DMATERIAL7

Call d3dDev.GetMaterial(mat)
If Err.Number <> DD_OK Then
' Code to handle the error goes here.
End If
```

Mathematics of Direct3D Lighting

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 backtransformed. As a result, if the world or

view matrices introduce non-uniform scaling, the resultant lighting could 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.

- Light Color Types and Sources
- Light Attenuation Over Distance
- Reflectance Model
- Spotlight Falloff Model
- Fog Effects

Light Color Types and Sources

Light sources in Direct3D emit diffuse, ambient, and specular colors as distinct light components that factor into lighting computations independently of each other. (Note that this is a departure from the light model employed in versions of DirectX prior to DirectX 7.0, which used a single light color for all lighting calculations.) 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 `D3DRENDERSTATE_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 `D3DRENDERSTATE_AMBIENTMATERIALSOURCE`, `D3DRENDERSTATE_DIFFUSEMATERIALSOURCE`, and `D3DRENDERSTATE_SPECULARMATERIALSOURCE` render states control the source from which the system draws the associated colors during lighting calculations. Each of these render states can be set 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 Visual Basic, vertex colors are used by the system only when you set the `D3DRENDERSTATE_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 `D3DRENDERSTATE_AMBIENTMATERIALSOURCE`, `D3DRENDERSTATE_DIFFUSEMATERIALSOURCE`, and

D3DRENDERSTATE_SPECULARMATERIALSOURCE render states control the source from which the system draws the associated colors during lighting calculations. Each of these render states can be set 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.

Light Attenuation Over Distance

[C++]

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:

$$D = \left\| V - L \right\|$$

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 (**dvRange**), no further attenuation calculations are made and no light effects from the light are applied 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 constant, linear, and quadratic attenuation factors as specified by the members of a light object's **D3DLIGHT7** structure. (Not surprisingly, the corresponding structure members are **dvAttenuation0**, **dvAttenuation1**, and **dvAttenuation2**.) The system normalizes D to be within the range [0.0,1.0], where 0.0 indicates no light at the vertex, and 1.0 indicates full light intensity at the vertex.

The constant, linear and quadratic attenuation factors act as coefficients in the formula—you can produce a wide variety of attenuation curves by making simple adjustments to them. You could set the constant attenuation factor to 1.0 to create a light that doesn't attenuate (but will still be limited by range), or you can experiment with different values to achieve various attenuation effects.

The attenuation formula used by Direct3D computes an attenuation value that typically ranges from 1.0 at the light source to 0.0 at the maximum range of the light. The attenuation value is multiplied into 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, as well as the reflectance of the current material to come up with the diffuse and specular components for that vertex. For more information, see Spotlight Falloff Model and Reflectance Model.

[Visual Basic]

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:

$$D = \left\| V - L \right\|$$

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 (**range**), no further attenuation calculations are made and no light effects from the light are applied 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):

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

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 constant, linear, and quadratic attenuation factors as specified by the members of a light object's **D3DLIGHT7** type. (Not surprisingly, the corresponding structure members are **attenuation0**, **attenuation1**, and **attenuation2**.) The system normalizes D to be within the range [0.0,1.0], where 0.0 indicates no light at the vertex, and 1.0 indicates full light intensity at the vertex.

The constant, linear and quadratic attenuation factors act as coefficients in the formula—you can produce a wide variety of attenuation curves by making simple adjustments to them. You could set the constant attenuation factor to 1.0 to create a light that doesn't attenuate (but will still be limited by range), or you can experiment with different values to achieve various attenuation effects.

The attenuation formula used by Direct3D computes an attenuation value that typically ranges from 1.0 at the light source to 0.0 at the maximum range of the light. The attenuation value is multiplied into 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, as well as the reflectance of the current material to come up with 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, 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 figuring out 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.

This section provides information on the methods that the system uses for calculating reflectance. Information is divided according to the type of reflectance being calculated:

- Diffuse Reflection Model
- Specular Reflection Model

Diffuse Reflection Model

[C++]

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, and vector N is normalized only if the D3DRENDERSTATE_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 $I_a V_a + V_e$ once, adding $A(R_d V_d L_d + V_a L_a)$ for every active light.

If the D3DRENDERSTATE_COLORVERTEX render state is enabled, the system selects colors for V based on the values set for the D3DRENDERSTATE_AMBIENTMATERIALSOURCE and D3DRENDERSTATE_DIFFUSEMATERIALSOURCE render states. Set these render states set 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]

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, and vector N is normalized only if the D3DRENDERSTATE_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 $I_a V_a + V_e$ once, adding $A(R_d V_d L_d + V_a L_a)$ for every active light.

If the D3DRENDERSTATE_COLORVERTEX render state is enabled, the system selects colors for V based on the values set for the

D3DRENDERSTATE_AMBIENTMATERIALSOURCE and D3DRENDERSTATE_DIFFUSEMATERIALSOURCE render states. Set these render states set 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. Direct3D provides applications with two ways to compute the halfway vector, controlled by the D3DRENDERSTATE_LOCALVIEWER render state. If D3DRENDERSTATE_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:

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

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 is somewhat computationally intensive. An alternative would be to set D3DRENDERSTATE_LOCALVIEWER to FALSE, which 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 D3DRENDERSTATE_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:

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

In the preceding formula, R_s is the specular reflectance, N is the vertex normal, H is the halfway vector, and p is the specular reflection power of the current material (as specified by the **dvPower** member of the material's **D3DMATERIAL7** structure). Vector H is normalized, and vector N is normalized only if the **D3DRENDERSTATE_NORMALIZENORMALS** render state is enabled.

Like 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 (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 **D3DRENDERSTATE_COLORVERTEX** render state is enabled, the system selects the color source for V based on the value of the **D3DRENDERSTATE_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. Direct3D provides applications with two ways to compute the halfway vector, controlled by the **D3DRENDERSTATE_LOCALVIEWER** render state. If **D3DRENDERSTATE_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:

$$H = \text{norm} \left(\text{norm} (V - L) \right)$$

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 is somewhat computationally intensive. An alternative would be to set `D3DRENDERSTATE_LOCALVIEWER` to `False`, which 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 `D3DRENDERSTATE_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 is 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 **D3DMATERIAL7** type). Vector H is normalized, and vector N is normalized only if the `D3DRENDERSTATE_NORMALIZENORMALS` render state is enabled.

Like 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 (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 `D3DRENDERSTATE_COLORVERTEX` render state is enabled, the system selects the color source for V based on the value of the `D3DRENDERSTATE_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.

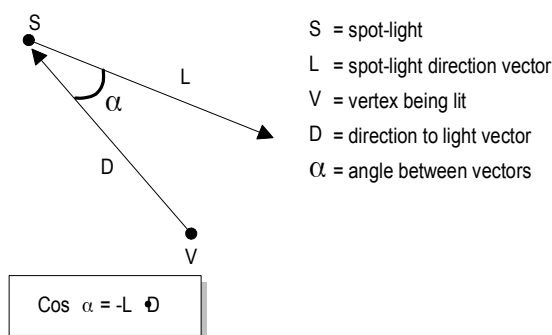
For more information, see Diffuse Reflection Model.

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 within 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 the light a vertex receives is based on the vertex's location within the inner or outer cones. 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:

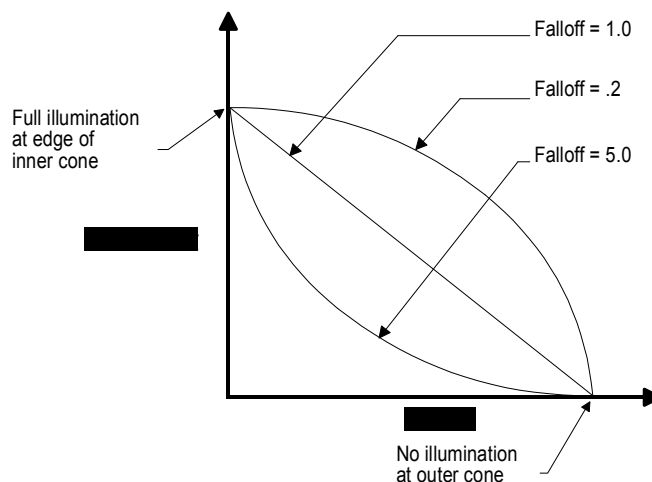


Next, the system compares this value to the cosine of the spotlight's inner and outer cone angles. In the light's **D3DLIGHT7** structure, the **dvTheta** and **dvPhi** members represent the total cone angles for the inner and outer cones. Because the attenuation occurs as the vertex becomes more distant 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 (**dvFalloff** in the **D3DLIGHT7** 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 **dvFalloff** member of the **D3DLIGHT7** structure and controls the shape of the falloff curve. The following illustration shows how different **dvFalloff** values can affect the falloff curve:



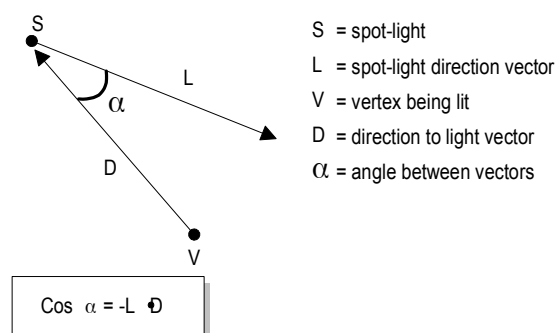
The effect of various **dvFalloff** values on the actual lighting is subtle, and a small performance penalty is incurred by shaping the falloff curve with **dvFalloff** values other than 1.0. For these reasons, most people will set this value to 1.0.

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 within 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 the light a vertex receives is based on the vertex's location within the inner or outer cones. 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:

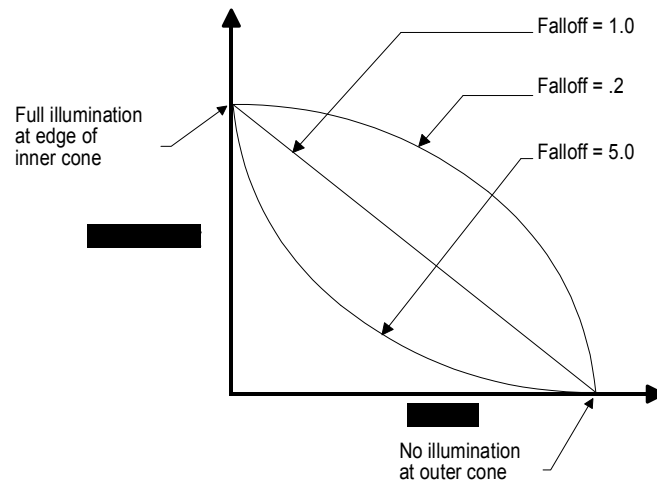


Next, the system compares this value to the cosine of the spotlight's inner and outer cone angles. In the light's **D3DLIGHT7** structure, the **theta** and **phi** members represent the total cone angles for the inner and outer cones. Because the attenuation occurs as the vertex becomes more distant 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:

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

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 **D3DLIGHT7** 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 **D3DLIGHT7** 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, most people will set this value to 1.0.

For more information, see [Light Attenuation Over Distance](#).

Fog Effects

The 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 into the alpha component of the vertex to be rasterized. For more information see [Fog and Vertex Fog](#).

Vertex Formats

This section describes the concepts you need to understand to specify vertices in 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
- Strided Vertex Format

[\[C++\]](#)

Note

Prior to DirectX 6.0, applications were required to use one of three vertex types—**D3DVERTEX**, **D3DLVERTEX**, and **D3DTLVERTEX**—depending on which parts of the Direct3D geometry pipeline were being used. With support for more flexible vertex formats in DirectX 6.0 and later, you can declare vertices in many more ways than before, but you can still use the predefined structures to describe untransformed and unlit vertices, untransformed but lit vertices, and vertices that are both transformed and lit. For more information, read the topics for each type of vertex in this section thoroughly.

[C++, Visual Basic]

About Vertex Formats

Direct3D Immediate Mode applications can define model vertices in several different ways. Support for flexible vertex definitions (also known as "flexible vertex formats") 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.

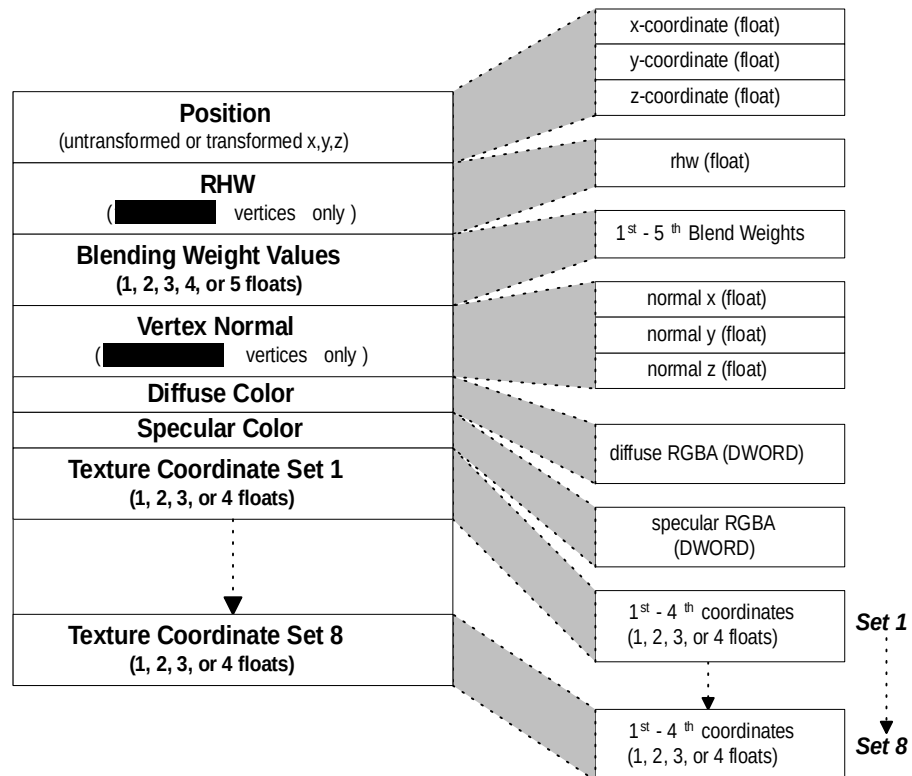
[C++]

The rendering methods of the **IDirect3DDevice7** interface presents 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.

Note

The behavior is slightly different if your application uses the **IDirect3DDevice7::DrawPrimitiveStrided** or **IDirect3DDevice7::DrawIndexedPrimitiveStrided** methods. For more information, see Strided Vertex Format.

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

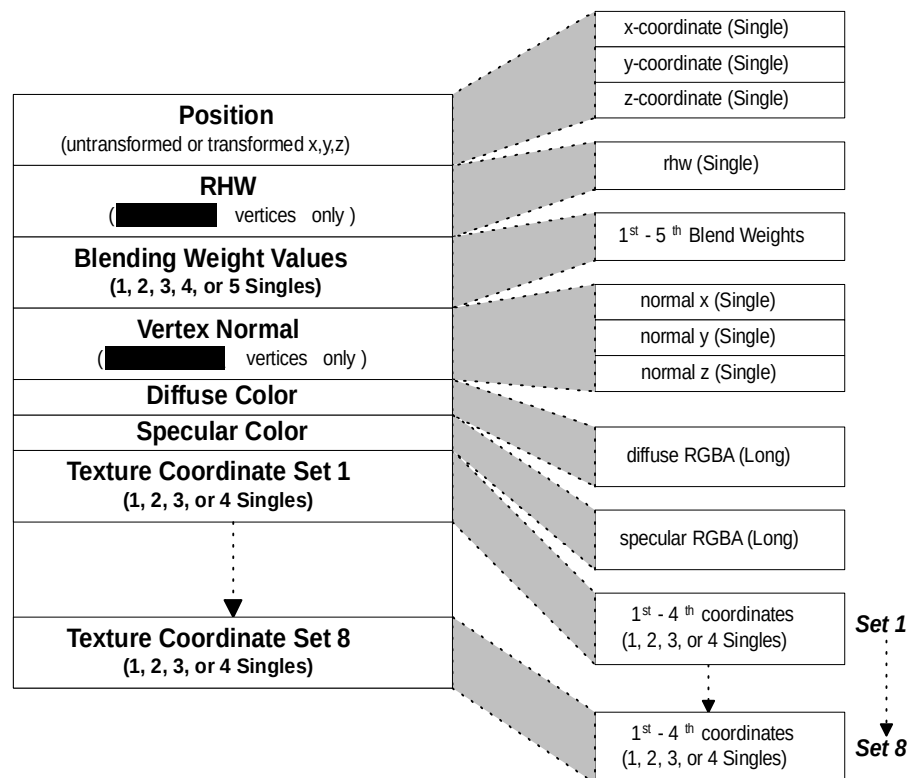
In DirectX 7.0, texture coordinates can be declared in different formats, allowing textures to be addressed using as few as one coordinate, or as many as three texture coordinates (for 2-D projected texture coordinates). For more information, see Texture Coordinate Formats. Use the **D3DFVF_TEXCOORDSIZE_n** set of macros to create bit patterns that identify the texture coordinate formats that your vertex format uses.

No real application will use every single component—the RHW (reciprocal homogenous W) 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 can't 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 flags in Flexible Vertex Format Flags.

[Visual Basic]

The **Direct3DDevice7** 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.

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 three texture coordinates (for 2-D projected texture coordinates). Use the **D3DFVF_TEXCOORDSIZEn** set of helper functions from the 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 RHW (reciprocal homogenous W) 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 can't 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 flags in Flexible Vertex Format Flags.

Untransformed and Unlit Vertices

The presence of the D3DFVF_XYZ (or any of the D3DFVF_XYZBn flags) 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 Direct3D perform all transformation and lighting operations using its internal algorithms. (If you want, you can disable the Direct3D lighting engine for the primitives being rendered by setting the D3DRENDERSTATE_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 (a model coordinate vertex is positioned relative to a local origin for the model, not for the world). The system then applies world, view, and projection transformations to the model coordinates to position them within your scene, and determine their final locations on the screen.
- 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, meaning that they 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, Lighting and 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, if you want to include a diffuse or specular color with your untransformed vertices, you can. (This wasn't possible before DirectX 6.0). Including individual colors for each vertex makes it possible to achieve shading effects that are much 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 D3DRENDERSTATE_COLORVERTEX render state. Untransformed, unlit vertices can also include up to eight sets of texture coordinates.

Applications can still use the legacy **D3DVERTEX** structure for vertices. In fact, the `d3dypes.h` header file defines a shortcut macro to identify this vertex format:

```
#define D3DFVF_VERTEX ( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1 )
```

If the **D3DVERTEX** structure doesn't 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 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 dwDiffuseRGBA; // diffuse color  
  
    float tu1, // texture coordinates  
    tv1;  
  
    float tu2,  
    tv2;  
} UNLITVERTEX, *LPUNLITVERTEX;
```

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. The rendering methods, such as

IDirect3DDevice7::DrawPrimitive, accept the address of a vertex array as a void pointer, so remember to cast your vertex array pointer to the **LPVOID** data type when you call the rendering methods.

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, if you want to include a diffuse or specular color with your untransformed vertices, you can. (This wasn't possible before DirectX 6.0). Including individual colors for each vertex makes it possible to achieve shading effects that are much 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 `D3DRENDERSTATE_COLORVERTEX` render state. Untransformed, unlit vertices

can also include up to eight sets of texture coordinates. Visual Basic applications can use the **D3DVERTEX** type for vertices.

If the **D3DVERTEX** type doesn't 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 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
```

```
  DiffuseRGBA 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. The rendering methods, such as

Direct3DDevice7.DrawPrimitive, accept the first element of a vertex array as type **Any**, to accommodate all types of vertex formats.

For more information, see About Vertex Formats.

Untransformed and Lit Vertices

If you include the D3DFVF_XYZ flag, but not the D3DFVF_NORMAL flag, in the vertex format description you use with the Direct3D rendering methods, you are identifying your vertices as untransformed, but already lit. (For information about other dependencies and exclusions, see 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 both the diffuse and specular colors is 0xFFFFFFFF.

Like the other vertex types, other than 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++]

C++ applications can still use the legacy **D3DLVERTEX** structure for vertices. The `d3dtypes.h` header file defines the following helper macro that you can use to describe the **D3DLVERTEX** structure's format:

```
#define D3DFVF_LVERTEX ( D3DFVF_XYZ | D3DFVF_RESERVED1 | D3DFVF_DIFFUSE | \
    D3DFVF_SPECULAR | D3DFVF_TEX1 )
```

Note that the helper macro includes the `D3DFVF_RESERVED1` flag, indicating to the system that you're using the **D3DLVERTEX** structure, which includes the **dwReserved** member. This is required when using **D3DLVERTEX** because vertex formats don't usually include reserved fields; the `D3DFVF_RESERVED1` flag informs the system that there is an unused **DWORD** between the vertex's position and diffuse color vertex components.

If the **D3DLVERTEX** structure doesn't include all the fields your application needs, you can define another structure. Make sure that your vertex components appear in the required order, declaring a new structure accordingly. The following code 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 dwDiffuseRGBA; // diffuse color

    DWORD dwSpecularRGBA; // specular color

    float tu1, // texture coordinates
    tv1;

    float tu2,
```

```
tv2;  
  
    float tu3,  
    tv3;  
} LITVERTEX, *LPLITVERTEX;
```

The vertex description for the preceding structure would be a combination of the D3DFVF_XYZ, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX3 flexible vertex format flags. The rendering methods, such as

IDirect3DDevice7::DrawPrimitive, accept the address of a vertex array as a void pointer, so remember to cast your vertex array pointer to the **LPVOID** data type when you call the rendering methods.

For more information, see About Vertex Formats.

[Visual Basic]

Applications written in Visual Basic can define their own vertex formats, or use the **D3DLVERTEX** type for lit vertices. If the **D3DLVERTEX** type doesn't 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 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
```

```
    DiffuseRGBA As Long ' diffuse color
```

```
    SpecularRGBA 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 would be a combination of the D3DFVF_XYZ, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX3 flexible vertex format flags. The rendering methods, such as **Direct3DDevice7.DrawPrimitive**, accept the first element of a vertex array as type **Any**, to accommodate all types of vertex formats.

For more information, see About Vertex Formats.

Transformed and Lit Vertices

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 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 already taken care of these steps. (This fact makes transformed and lit vertices common when porting existing 3-D applications to Direct3D Immediate Mode.) 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 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 viewer, and 1.0 is the farthest position still visible within the viewing area. Immediately following the position, transformed and lit vertices must include an RHW value (reciprocal of homogeneous W) 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 0xFFFFFFFF for both these components.
- You can use up to eight sets of texture coordinates, or none at all.

[C++]

Applications written in C++ can still use the legacy **D3DTLVERTEX** structure for transformed and lit vertices. The d3dtypes.h header file defines the following helper

macro that you can use to describe the vertex format declared by the **D3DTLVERTEX** structure:

```
#define D3DFVF_TLVERTEX ( D3DFVF_XYZRHW | D3DFVF_DIFFUSE |  
D3DFVF_SPECULAR | \  
D3DFVF_TEX1 )
```

It's possible that the **D3DTLVERTEX** structure doesn't include the fields you need. If this is the case, define another structure that does, but make sure that the vertex components are ordered properly. The following code 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 dwDiffuseRGBA; // diffuse color  
  
    DWORD dwSpecularRGBA; // specular color  
  
    float tu1, // texture coordinates  
    tv1;  
} 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. The rendering methods, such as **IDirect3DDevice7::DrawPrimitive**, accept the address of a vertex array as a void pointer, so remember to cast your vertex array pointer to the **LPVOID** data type when you call the rendering methods.

For more information, see About Vertex Formats.

[\[Visual Basic\]](#)

Visual Basic applications can use the **D3DTLVERTEX** type for transformed and lit vertices. It's possible that the **D3DTLVERTEX** type doesn't include the fields you need. If this is the case, define another type that does, but make sure that the vertex components are ordered properly. The following code 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
```

```
    DiffuseRGBA As Long ' diffuse color
```

```
    dwSpecularRGBA As Long ' specular color
```

```
    tu1 As Single ' texture coordinates
```

```
    tv1 As Single
```

```
End Type
```

The vertex description for the preceding type would be a combination of the D3DFVF_XYZRHW, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX1 flexible vertex format flags. The rendering methods, such as **Direct3DDevice7.DrawPrimitive**, accept the first element of a vertex array as type **Any**, to accommodate all types of vertex formats.

For more information, see About Vertex Formats.

Strided Vertex Format

[\[Visual Basic\]](#)

Note

Information in this topic pertains only to applications written in C++. DirectX for Visual Basic does not support strided vertices.

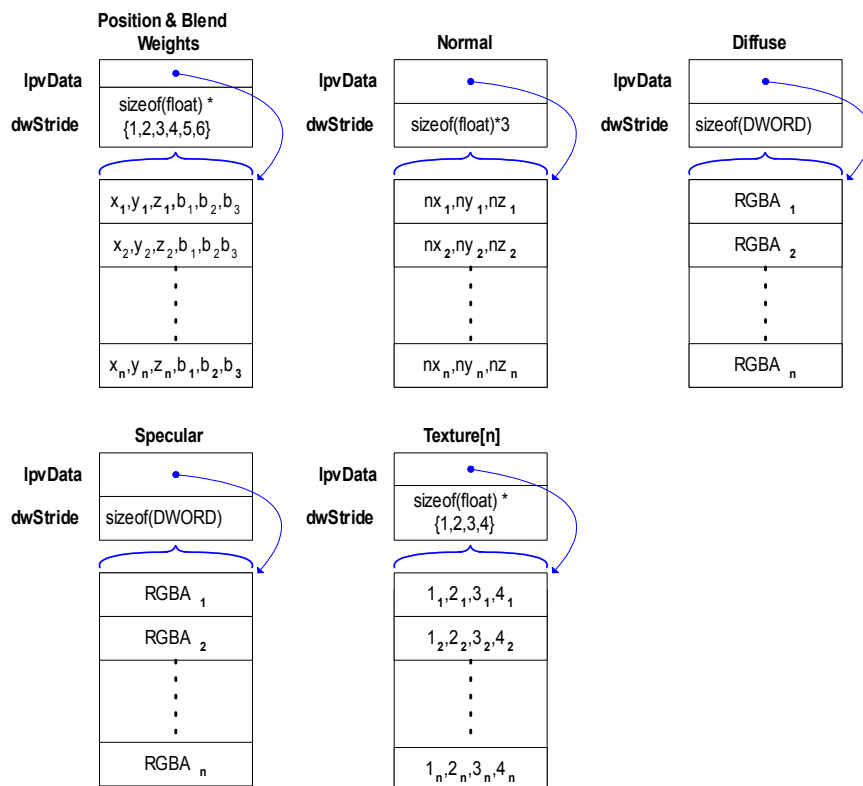
[\[C++\]](#)

The strided vertex format contains fields to represent untransformed vertices, lit or unlit, for the **IDirect3DDevice7::DrawPrimitiveStrided**, **IDirect3DDevice7::DrawIndexedPrimitiveStrided**, and **IDirect3DVertexBuffer7::ProcessVerticesStrided** methods. You cannot use the strided vertex format for transformed vertices. Unlike a "normal" vertex, which is a structure that physically contains all the necessary vertex components, a "strided"

vertex is a structure that contains pointers to the vertex components rather than the components themselves.

This indirection is accomplished through the **D3DDRAWPRIMITIVESTRIDEEDDATA** structure that is accepted by the **DrawPrimitiveStrided**, **DrawIndexedPrimitiveStrided**, and **ProcessVerticesStrided** methods. You describe strided vertices using the same combinations of flexible vertex format flags as a non-strided vertex. However, unlike non-strided vertices, the flags you use do not indicate the presence of a given field in memory (**D3DDRAWPRIMITIVESTRIDEEDDATA** includes them all), they indicate which the structure members that your application uses. The restrictions for these flags are identical to non-strided vertices, for details, see Flexible Vertex Format Flags.

The **D3DDRAWPRIMITIVESTRIDEEDDATA** structure contains 12 **D3DDP_PTRSTRIDE** structures, one structure each for the position, normal, diffuse color, specular color, and texture coordinate sets for the vertices. Each **D3DDP_PTRSTRIDE** structure contains a pointer to an array of data, and the stride of that array. These 12 structures provide the indirection that allows your application to arrange vertex components however it needs. For instance, you might use distinct arrays for every vertex component, as shown in the following illustration.



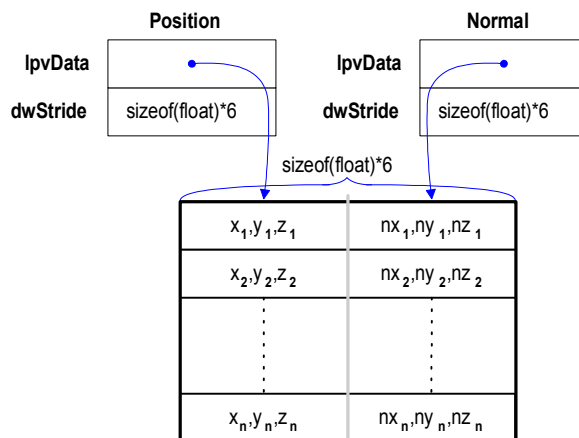
The **lpvData** member of the corresponding **D3DDP_PTRSTRIDE** structures contains the address of a buffer that contains an array of vertex components. Each element in each array represents a component for one vertex, which is made up of some number of floats (for position, normal, and texture coordinates) or an RGBA value (for diffuse and specular colors). There is one entry within each array for every vertex to be rendered. The **dwStride** member of **D3DDP_PTRSTRIDE** should be set to the memory stride, in bytes, from one entry in the array to the next. The following table describes the components and the corresponding strides for each.

Component	Stride
untransformed position	3 to 6 floats (x,y,z, plus 0 to 3 blend weights)
vertex normal	3 floats (nx,ny,nz)
diffuse color	1 DWORD (RGBA)
specular color	1 DWORD (RGBA)
texture coordinate set	1, 2, 3, or 4, floats (1st, 2nd, 3rd, or 4th texture coordinates)

Note

In DirectX 7.0, texture coordinates can be declared in different formats, allowing textures to be addressed using as few as one coordinate, or as many as three 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.

Some developers might choose to include some or all vertex components in an interleaved array. The strided vertex format makes doing this very simple. If your application interleaves the vertex position and normal components, for example, you could visualize the memory layout and corresponding settings in **D3DDP_PTRSTRIDE** structures as follows.



In this case, the **lpvData** members of the **D3DDP_PTRSTRIDE** structures point to two locations within the same buffer, and the strides are set to the combined width of the interleaved components. Of course, you aren't limited to any particular interleaving scheme. So long as you set the data pointers and their strides correctly, any interleaving scheme will work.

Textures

Textures are a powerful tool in the quest for realism in computer-generated 3-D images. 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 Surface Objects
- Texture Filtering
- Texture Wrapping
- Texture Blending
- Texture Compression
- Automatic Texture Management
- Hardware Considerations for Texturing

If you are familiar with what textures are and how they are used, you may want to skip the Basic Texturing Concepts section. It is important to note, however, that Direct3D now uses texture interfaces to access many of its most powerful texturing features. Therefore, you may want to look over the section entitled Texture Surface Objects.

Basic Texturing Concepts

This section presents the most fundamental concepts required for an understanding of texturing in Direct3D. The information in this section is presented in the following topics:

- What Is a Texture?
- Texture Addressing Modes
- Texture Interfaces and Texture Handles
- Palettized Textures

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 that give 3-D objects realistic visual complexity such as scuffs, cracks, fingerprints, and smudges. In recent years, textures have gained popularity among developers as a tool for enhancing the realism of computer-generated 3-D images.

At its most basic, a texture is simply a bitmap of pixel colors. In this sense, the word texture has a specific definition when used in the context of computer graphics. In the normal semantics associated with the word texture, we refer both to the patterns of color on an object and its roughness or smoothness. Direct3D textures don't add "bumpiness" to an object. Rather, the textures, or patterns of colors, just give it the appearance of bumpiness.

Because Direct3D textures are simply 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 very realistic-looking hillside. Texturing can also be used 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. Information on these techniques is presented in [Texture Blending and Light Mapping With Textures](#).

If your application creates a HAL device, an MMX device, or an RGB device (see [Direct3D Device Types](#)), it can use 8-, 16-, 24-, or 32-bit textures. Legacy applications that use the monochromatic (or ramp) device must use 8-bit textures.

Texture Addressing Modes

This section describes the purpose and use of Direct3D texture addressing modes. It is organized into the following topics:

- [What Are Texture Addressing Modes?](#)
- [About the Wrap Texture Address Mode](#)
- [About the Mirror Texture Address Mode](#)
- [About the Clamp Texture Address Mode](#)
- [About the 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 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 will be 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 such that a texture is tiled across a primitive. The following topics contain additional details:

- About the Wrap Texture Address Mode
- About the Mirror Texture Address Mode
- About the Clamp Texture Address Mode
- About the Border Color Texture Address Mode

About the Wrap Texture Address Mode

[C++]

The "wrap" texture address mode, identified by the `D3DTADDRESS_WRAP` member of the **D3DTEXTUREADDRESS** enumerated type, makes Direct3D repeat the texture on every integer junction. Suppose, for example, your program 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` will result 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 Direct3D repeat the texture on every integer junction. Suppose, for example, your program 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` will result 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 About the Mirror Texture Address Mode.

About the Mirror Texture Address Mode

[C++]

The "mirror" texture address mode, identified by the `D3DTADDRESS_MIRROR` member of the **D3DTEXTUREADDRESS** enumerated type, causes Direct3D to mirror the texture at every integer boundary. Suppose, for example, your program 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` will result in the texture being applied three times in both the u- and v-directions. Every other row and column that it is applied to will be 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 Direct3D to mirror the texture at every integer boundary. Suppose, for example, your program 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` will result in the texture being applied three times in both the u- and v-directions. Every other row and column that it is applied to will be 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 About the Wrap Texture Address Mode.

About the Clamp Texture Address Mode

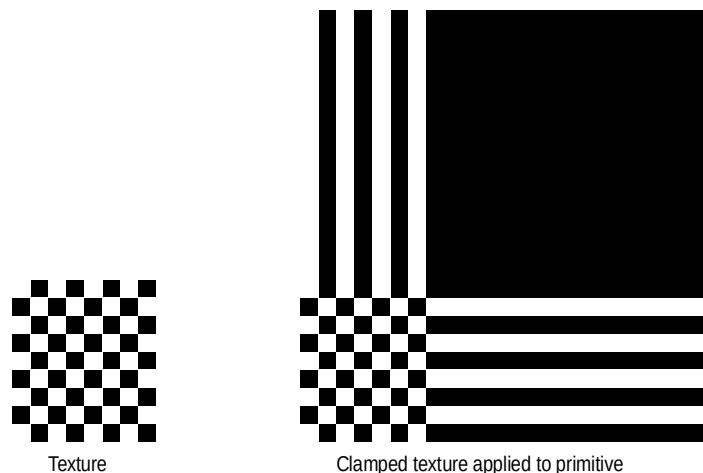
[C++]

The "clamp" texture address mode, identified by the `D3DTADDRESS_CLAMP` member of the **D3DTEXTUREADDRESS** enumerated type, causes Direct3D to clamp your texture coordinates to the $[0.0, 1.0]$ range. That is, it will apply the texture once, then "smear" the color of edge pixels. For instance, suppose that your program 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` will result 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 Direct3D to clamp your texture coordinates to the $[0.0, 1.0]$ range. That is, it will apply the texture once, then "smear" the color of edge pixels. For instance, suppose that your program 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` will result 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:



About the Border Color Texture Address Mode

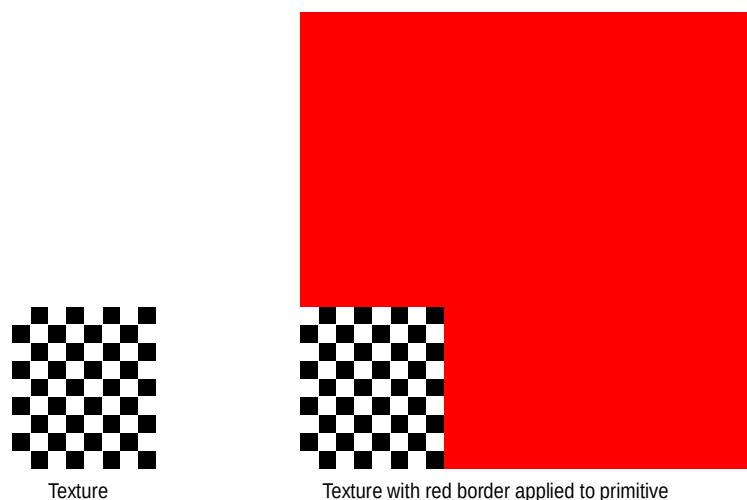
[C++]

The "border color" texture address mode, identified by the `D3DTADDRESS_BORDER` member of the **D3DTEXTUREADDRESS** enumerated type, causes 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 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 next illustration in which the application specified that the texture be applied to the primitive using a red border.



[C++]

Applications set the border color by calling **IDirect3DDevice7::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.

If your application stills uses the legacy **IDirect3DDevice2** interface, you can set the border color by calling **IDirect3DDevice2::SetRenderState** method, specifying the **D3DRENDERSTATE_BORDERCOLOR** render state value and the new RGBA border color as parameters. (The **D3DRENDERSTATE_BORDERCOLOR** render state is obsolete and is not recognized by the **IDirect3DDevice7** interface.)

[Visual Basic]

Visual Basic applications set the border color by calling **Direct3DDevice7.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 **IDirect3DDevice7::SetTextureStageState** method. Specify the desired texture stage identifier in the first parameter. Set the second parameter to **D3DTSS_ADDRESS** to change both the u- and v-texture addressing modes simultaneously, or use the **D3DTSS_ADDRESSU** or **D3DTSS_ADDRESSV** values to update the u- or v-

addressing modes individually. The third parameter you pass to **SetTextureStageState** determines which mode is being set; this can be any one of the members of the **D3DTEXTUREADDRESS** enumerated type. To retrieve the current texture address mode for a given texture stage, call **IDirect3DDevice7::GetTextureStageState**, using the **D3DTSS_ADDRESS**, **D3DTSS_ADDRESSU**, or **D3DTSS_ADDRESSV** members of the **D3DTEXTURESTAGESTATETYPE** enumeration to identify the address mode about which you want information.

If your application uses the legacy **IDirect3DDevice2** interface, you set texture addressing modes by calling the **IDirect3DDevice2::SetRenderState** method, using the **D3DRENDERSTATE_TEXTUREADDRESS** render state to simultaneously set u- and v-texture addressing. You can set u- and v-addressing individually with the **D3DRENDERSTATE_TEXTUREADDRESSU** or **D3DRENDERSTATE_TEXTUREADDRESSV** render states. Like their newer **SetTextureStageState** counterparts, these render states also use the values from **D3DTEXTUREADDRESS** enumerated type.

Note

The **IDirect3DDevice7::SetRenderState** method (as opposed to the **IDirect3DDevice2** version) still recognizes the **D3DRENDERSTATE_TEXTUREADDRESS**, **D3DRENDERSTATE_TEXTUREADDRESSU**, **D3DRENDERSTATE_TEXTUREADDRESSV** render states, even though they have been superseded. Instead of failing these legacy render states, the **IDirect3DDevice7** implementation maps their effects to the first texture stage (stage 0). Applications should not mix the legacy render states with their corresponding texture stage states, as unpredictable results can occur.

[\[Visual Basic\]](#)

From Visual Basic, you set texture addressing modes for individual texture stages by calling the **Direct3DDevice7.SetTextureStageState** method. Specify the desired texture stage identifier in the first parameter. Set the second parameter to **D3DTSS_ADDRESS** to change both the u- and v-texture addressing modes simultaneously, or use the **D3DTSS_ADDRESSU** or **D3DTSS_ADDRESSV** values to update the u- or v-addressing modes individually. The third parameter you pass to **SetTextureStageState** determines which mode is being set; this can be any one of the members of the **CONST_D3DTEXTUREADDRESS** enumeration.

To retrieve the current texture address mode for a given texture stage, call **Direct3DDevice7.GetTextureStageState**, using the **D3DTSS_ADDRESS**, **D3DTSS_ADDRESSU**, or **D3DTSS_ADDRESSV** 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 **`dwMaxTextureRepeat`** member of the **`D3DDEVICEDESC7`** 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 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 **`dwMaxTextureRepeat`** is also affected by the `D3DPTTEXTURECAPS_TEXREPEATNOTSCALEDDBYSIZE` capability bit. When this bit is set, the value in the **`dwMaxTextureRepeat`** member of **`D3DDEVICEDESC7`** is used precisely as described. However, when `D3DPTTEXTURECAPS_TEXREPEATNOTSCALEDDBYSIZE` is not set, texture repeating limitations depend on the size of the texture indexed by the texture coordinates. In this case, **`dwMaxTextureRepeat`** 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 **`dwMaxTextureRepeat`** 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 **`IMaxTextureRepeat`** member of the **`D3DDEVICEDESC7`** 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 **IMaxTextureRepeat** is also affected by the **D3DPTTEXTURECAPS_TEXREPEATNOTSCALEDDBYSIZE** capability bit. When this bit is set, the value in the **IMaxTextureRepeat** member of **D3DDEVICEDESC7** is used precisely as described. However, when **D3DPTTEXTURECAPS_TEXREPEATNOTSCALEDDBYSIZE** is not set, texture repeating limitations depend on the size of the texture indexed by the texture coordinates. In this case, **IMaxTextureRepeat** 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 **IMaxTextureRepeat** 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 Interfaces and Texture Handles

[\[Visual Basic\]](#)

Note

Information in this topic pertains only to applications written in C++. DirectX for Visual Basic does not use texture handles.

[\[C++\]](#)

Texture handles are obsolete. They were used in applications that utilized the **IDirect3D** and **IDirect3D2** interfaces.

With the **IDirect3D7** interface (and the legacy **IDirect3D3** interface), you create and use textures through interface pointers to the texture surfaces. You obtain a texture surface interface pointer when you create the texture surface by calling the **IDirectDraw7::CreateSurface** method. DirectX3D supports the blending of up to eight textures onto a primitive at once. For details, see Texture Surface Objects.

Palettized Textures

[\[C++\]](#)

Direct3D devices can support texturing from texture surfaces that use attached palettes. These types of textures are sometimes called "palettized textures." A palettized texture is a **DirectDrawSurface** object, created with the **DDSCAPS_TEXTURE** capability, that uses one of the **DDPF_PALETTEINDEXED n** pixel formats (where n is 1, 2, 4, or 8). These

capabilities are included in the **DDSURFACEDESC2** structure that you use to create a texture (and the **DDPIXELFORMAT** structure that the description contains). Like all palettized surfaces, instead of a color, each pixel is an index into a table of values held within an attached **DirectDrawPalette** object. For more information about creating and using palettized surfaces, see *Surfaces and Palettes* in the *DirectDraw* documentation. As you should always do when checking texture-related device capabilities, be sure to verify the supported texture pixel formats by calling the **IDirect3DDevice7::EnumTextureFormats** method.

U To prepare a palettized texture

1. Check **DirectDraw** and **Direct3D** capabilities as described in this section.
2. Create a surface of the desired dimensions that includes the **DDSCAPS_TEXTURE** capability and uses one of the **DDPF_PALETTEINDEXED n** pixel formats.
3. Create and initialize a **DirectDrawPalette** object. For more information, see *Palettes* in the *DirectDraw* documentation. (To use an alpha-only palettized texture, include the **DDPCAPS_ALPHA** capability when you create the palette. See the reference for **IDirectDraw7::CreatePalette** for more information.)
4. Attach the palette to the surface by calling the **IDirectDrawSurface7::SetPalette** method for the texture surface.

Note

If you create a palettized texture surface, but neglect to attach a palette, your application will cause an access violation within **Direct3D Immediate Mode** during rendering.

The palettes you use with palettized textures need not be limited to color data. In some cases, devices support texture palettes that also contain alpha information. If so, **DirectDraw** will expose the **DDPCAPS_ALPHA** palette capability flag when you call the **IDirectDraw7::GetCaps** method—the flag is found in the **dwPalCaps** member of the associated **DDCAPS** structure. If the rendering device can perform texturing from alpha-capable palettized textures, it will expose the **D3DPTTEXTURECAPS_ALPHAPALETTE** capability when you call **IDirect3DDevice7::GetCaps**. (The **D3DPTTEXTURECAPS_ALPHAPALETTE** flag can be found in the two **D3DPRIMCAPS** structures contained by the **D3DDEVICEDESC7** structure you pass with the call.)

[\[Visual Basic\]](#)

Direct3D devices can support texturing from texture surfaces that use attached palettes. These types of textures are sometimes called "palettized textures." A palettized texture is a **DirectDrawSurface** object, created with the **DDSCAPS_TEXTURE** capability, that uses one of the **DDPF_PALETTEINDEXED n** pixel formats (where n is 1, 2, 4, or 8). These capabilities are included in the **DDSURFACEDESC2** type that you use to create a texture (and the **DDPIXELFORMAT** type that the description contains). Like all

palettized surfaces, instead of a color, each pixel is an index into a table of values held within an attached `DirectDrawPalette` object. For more information about creating and using palettized surfaces, see *Surfaces and Palettes* in the *DirectDraw* documentation. As you should always do when checking texture-related device capabilities, be sure to verify the supported texture pixel formats by using `Direct3DEnumPixelFormats` class retrieved by the `Direct3DDevice7.GetTextureFormatsEnum` method.

U To prepare a palettized texture

1. Check *DirectDraw* and *Direct3D* capabilities as described in this section.
2. Create a surface of the desired dimensions that includes the `DDSCAPS_TEXTURE` capability and uses one of the `DDPF_PALETTEINDEXEDn` pixel formats.
3. Create and initialize a `DirectDrawPalette` object. For more information, see *Palettes* in the *DirectDraw* documentation. (To use an alpha-only palettized texture, include the `DDPCAPS_ALPHA` capability when you create the palette. See the reference for `DirectDraw7.CreatePalette` for more information.)
4. Attach the palette to the surface by calling the `DirectDrawSurface7.SetPalette` method for the texture surface.

Note

If you create a palettized texture surface, but neglect to attach a palette, your application will cause an access violation within *Direct3D Immediate Mode* during rendering.

The palettes you use with palettized textures need not be limited to color data. In some cases, devices support texture palettes that also contain alpha information. If so, *DirectDraw* will expose the `DDPCAPS_ALPHA` palette capability flag when you call the `DirectDraw7.GetCaps` method—the flag is found in the `IPalCaps` member of the associated `DDCAPS` type. If the rendering device can perform texturing from alpha-capable palettized textures, it will expose the `D3DPTTEXTURECAPS_ALPHAPALETTE` capability when you call `Direct3DDevice7.GetCaps`. (The `D3DPTTEXTURECAPS_ALPHAPALETTE` flag can be found in the two `D3DPRIMCAPS` types contained by the `D3DDEVICEDESC7` type you pass with the call.)

Texture Coordinates

The following topics introduce the concept of texture coordinates, their formats, and the *Direct3D* services used to manipulate them.

- Understanding Texture Coordinates
- Directly Mapping Texels to Pixels
- Texture Coordinate Formats

- Texture Coordinate Processing

Understanding Texture Coordinates

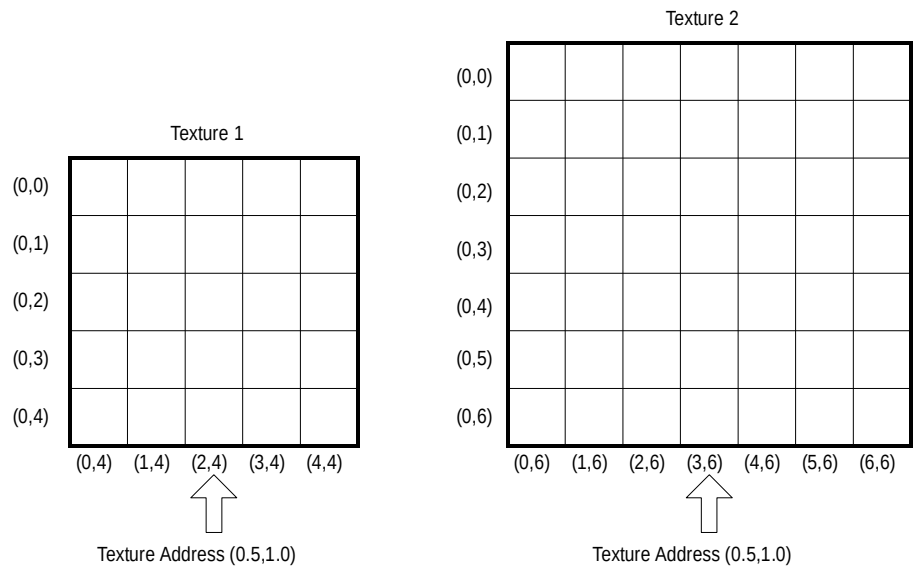
Most textures, like bitmaps, are a two dimensional array of color values. (Cubic-environment map textures are an exception. For details, see Cubic Environment Mapping.) The individual color values are called texture elements, or texels. Each texel has a unique address in the texture. The address can 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.

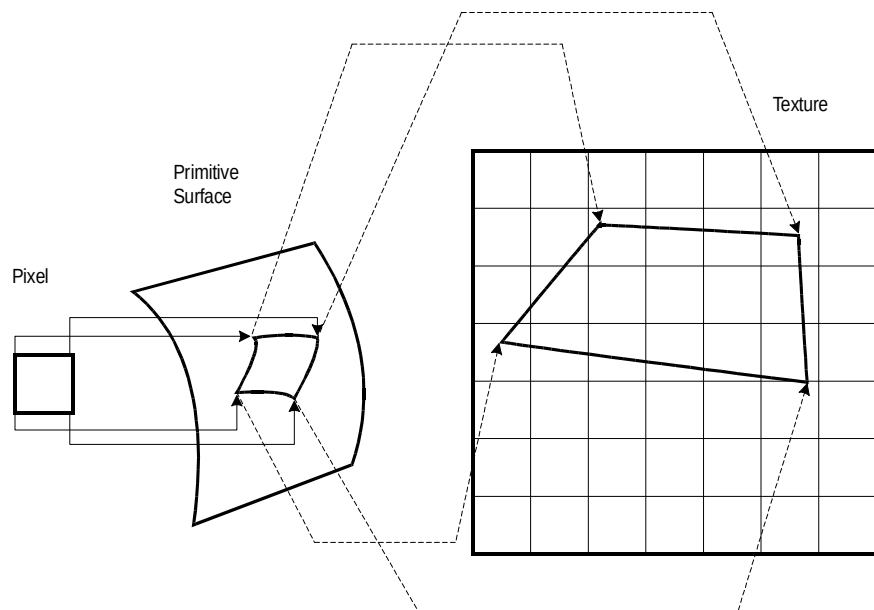
Direct3D maps texels in texture space directly to pixels in screen space, skipping the intermediate step for greater efficiency. This mapping process is actually an inverse mapping. That is, for each pixel in screen space, the corresponding texel position in texture space is calculated. The texture color 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 programs 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).



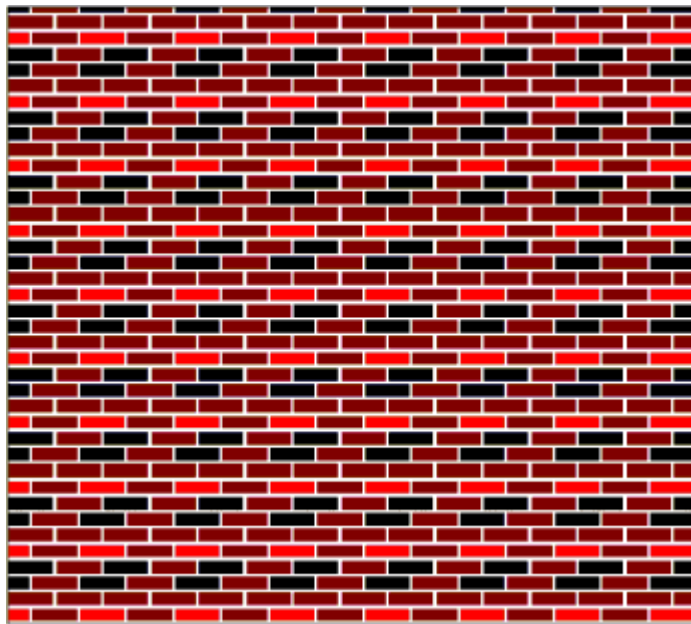
A simplified version of the texel mapping process is shown in the following diagram. (Admittedly, this example is extremely simple. For more detailed information, see [Directly Mapping Texels to Pixels](#).)



For this example, we are idealizing a pixel, shown at the left of the illustration, 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. For details, see the reference for the **D3DVERTEX** C++ structure (or the **D3DVERTEX** Visual Basic type). 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 would assign to the vertices of the primitive are (0.0,0.0), (1.0,0.0), (1.0,1.0), and (0.0,1.0).



Let's say you decide to decrease the height of the wall by one-half. You can either distort the texture to fit onto the smaller wall, or you can assign texture coordinates that will 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 would assign 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 will apply 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 such 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 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.

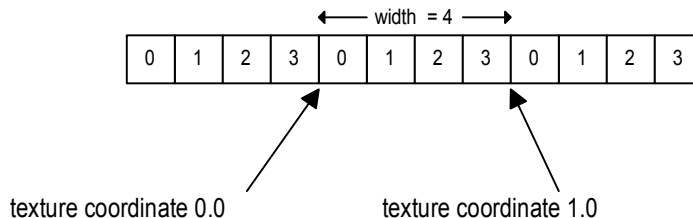
Direct3D performs the following computations to map floating point texture coordinates into texel addresses is.

$$T_x = (u \times M_x) - 0.5$$

$$T_y = (v \times M_y) - 0.5$$

In the preceding formulas, T_x and T_y are the horizontal and vertical output texel coordinates, and u , 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.)

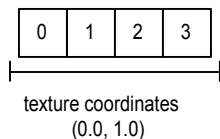
Placing the texture coordinate limits 0.0 and 1.0 into these formulas maps a texture coordinate of 0.0 halfway between the first and last texels of a repeated texture map. The coordinate 1.0 is mapped halfway between the last texel of the current iteration of the texture map and the next iteration. For 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.



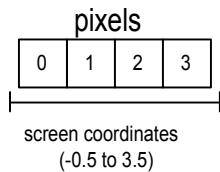
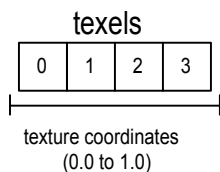
Given an understanding of this mapping, you can apply a simple bias to your screen-space geometry coordinates to force the system to map each texel to a corresponding pixel. For example, to draw a four-sided polygon (or "quad") that maps each texel from the preceding texture to one—and only one—pixel 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, you would draw a four-sided polygon (from two triangles) that have screen-space coordinates of -0.5 to 3.5, and texture coordinates of 0.0 to 1.0, respectively. 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 (remember the formulas). Scaling this by the texture size, which is 4, gives 0.5. Subtracting the 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. Given a texture of size 4, the mapping is as follows:



The system normalizes pixel coordinates in the same way it does texels, so if the vertices overlap the pixels into which you need to render, and the vertices use texture coordinates of 0.0 and 1.0, the two line up. If both are of similar size and properly aligned, they will match up exactly, texel to pixel, as shown in the following figure.



Texture Coordinate Formats

Texture coordinates in Direct3D can include 1, 2, 3, or 4 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) would be 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 within 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 1 and 4 elements. The D3DFVF_TEXTUREFORMAT1 through D3DFVF_TEXTUREFORMAT4 flags describe the number of elements in a given texture coordinate in a given 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 within 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 1 and 4 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 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.

```
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:  
' IFVF = 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.

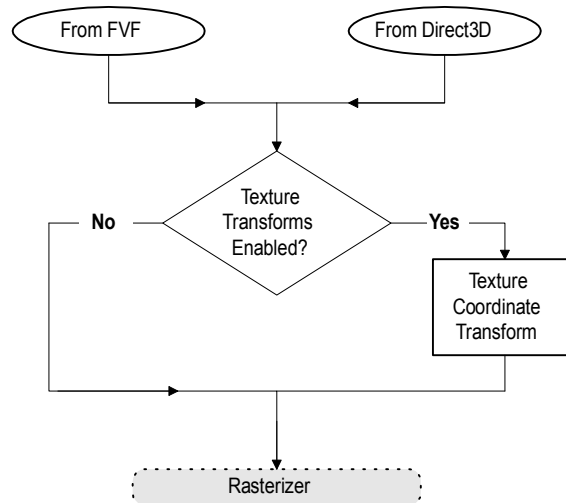
Texture Coordinate Processing

This section describes 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 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 1, 2, 3, or 4 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 one of the other members of **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 could 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 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 1, 2, 3, or 4 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 one of the other members 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 could 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 will be ignored. This also applies when using 2-D texture coordinates for a 1-D texture.

Automatically-Generated Texture Coordinates

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 normal from a vertex as texture coordinates, or it can compute the 3 element vectors used to address a cubic environment map. Like texture coordinates that you explicitly specify in a vertex, automatically-generated texture coordinates can be used 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 in concert with transformations to produce special effects. Of course, this feature is special-purpose, and there will be many occasions in which you will want to use explicit texture coordinates.

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 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 you assign to this state, the system's 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 as follows:

`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, so they cannot be combined. If you include one of these flags, you can still specify an index value, which the system uses to determine the texture wrapping mode.

[C++]

The following code fragment shows how these flags are used in C++.

```
// For this example, the lpD3DDevice variable is a valid
// pointer to an IDirect3DDevice7 interface.

// Use the vertex position (camera-space) as the input
// texture coordinates for this texture stage, and to use the
// wrap mode set in the D3DRENDERSTATE_WRAP1 render state
lpD3DDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX,
                                   D3DTSS_TCI_CAMERASPACEPOSITION | 1 );
```

[Visual Basic]

The following code fragment shows how these flags are used in Visual Basic.

```
' For this example, the d3dDevice variable is a valid
' reference to a Direct3DDevice7 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 3-element vectors for cubic-environment maps.

See Also

Texture Coordinate Transformations, Cubic Environment Mapping

Texture Coordinate Transformations

This following topics provide information about working with the texture coordinate transformation features offered by Direct3D Immediate Mode.

- About Texture Coordinate Transformations
- Setting and Retrieving Texture Coordinate Transformations
- Enabling Texture Coordinate Transformations

About Texture Coordinate Transformations

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 concatenation of these—can be applied.

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 might 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 **IDirect3DDevice7::SetTransform** and **IDirect3DDevice7::GetTransform** methods. These methods accept the D3DTRANSFORMSTATE_TEXTURE0 through D3DTRANSFORMSTATE_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 be applied to the texture coordinates for texture stage 0.

```
// For this example, the lpd3dDevice variable contains a
// valid pointer to an IDirect3DDevice7 interface.

D3DMATRIX matTrans = D3DUtil_SetIdentityMatrix(); // declared in d3dutil.h

// Set-up the matrix for the desired transformation.
lpd3dDevice->SetTransform(D3DTRANSFORMSTATE_TEXTURE0, &matTrans);
```

[Visual Basic]

Like the matrices that your application uses for geometry, you set and retrieve texture coordinate transformations by calling the **Direct3DDevice7.SetTransform** and **Direct3DDevice7.GetTransform** methods. These methods accept the D3DTRANSFORMSTATE_TEXTURE0 through D3DTRANSFORMSTATE_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 be applied to the texture coordinates for texture stage 0.

```
' For this example, the d3dDevice variable contains a
' valid reference to a Direct3DDevice7 object, and
' dx contains a valid reference to a DirectX7 object.
Dim matTrans As D3DMATRIX
Call dx.IdentityMatrix(matTrans)

' Set-up the matrix for the desired transformation.
Call d3dDevice.SetTransform(D3DTRANSFORMSTATE_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 lpD3DDevice variable contains a
// valid pointer to an IDirect3DDevice7 interface.

lpD3DDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTFF_DISABLE);
```

The other values defined in **D3DTEXTURETRANSFORMFLAGS** are used to enable texture coordinate transformations, and to control how many of the resulting texture coordinate elements are to be passed to the rasterizer. For example, take the following code:

```
// For this example, the lpD3DDevice variable contains a
// valid pointer to an IDirect3DDevice7 interface.

lpD3DDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTFF_COUNT2);
```

The D3DTTFF_COUNT2 value instructs the system to apply the transformation matrix set for texture stage 0, 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 lpD3DDevice variable contains a  
// valid pointer to an IDirect3DDevice7 interface.
```

```
lpD3DDevice->SetTextureStageState( 0, D3DTSS_TEXTURETRANSFORMFLAGS,  
                                   D3DTTFF_COUNT3 | D3DTTFF_PROJECTED );
```

The preceding code informs the system to pass three texture coordinate elements to the rasterizer, and the rasterizer is to divide 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 disables them:

```
' For this example, the d3DDevice variable contains a  
' valid reference to a Direct3DDevice7 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 of the resulting texture coordinate elements are to be passed to the rasterizer. For example, take the following code:

```
' For this example, the d3DDevice variable contains a  
' valid reference to a Direct3DDevice7 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, 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 IDirect3DDevice7 interface.

```
Call d3dDevice.SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, _
                                D3DTTFF_COUNT3 | D3DTTFF_PROJECTED)
```

The preceding code informs the system to pass three texture coordinate elements to the rasterizer, and the rasterizer is to divide first two elements by the third, producing the 2-D texture coordinates needed to address the texture.

Using Texture Coordinate Processing

[C++]

The following are some ways you might use texture coordinate processing to achieve special texturing effects, expressed with psuedo-code.

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 = D3DUtil_SetIdentityMatrix(); // 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!!!)
// 3 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 will be used.
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2)
• Define and set a matrix that applies a linear function:
// Generate texture coordinates as linear functions
// such that:
//   u = Ux*x + Uy*y + Uz*z + Uw
//   v = Vx*x + Vy*y + Vz*z + 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(D3DTRANSFORMSTATE_TEXTURE0, &M)

```

Performing environment mapping with a cubic environment map:

- Use the D3DTSS_TCI_CAMERASPACE REFLECTIONVECTOR flag to instruct the system to automatically generate texture coordinates (as reflection vectors) for cubic mapping.
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACE REFLECTIONVECTOR)
- Instruct the rasterizer to expect texture coordinates with 3 elements:
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT3)

Performing projective texturing:

- Use the D3DTSS_TCI_CAMERASPACE POSITION flag to instruct the system to pass the vertex position as input to a texture transformation matrix:
SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
D3DTSS_TCI_CAMERASPACE POSITION)
 - 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 3-element projected texture coordinates.
// Two output coordinates will be used.
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
D3DTTF_PROJECTED | D3DTTFF_COUNT3)
-

[\[Visual Basic\]](#)

The following are some ways you might use texture coordinate processing to achieve special texturing effects, expressed with psuedo-code.

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.
 - IFVFTex = (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!!!)
 - ' 3 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 will be used.
 - SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS, D3DTTFF_COUNT2)
- Define and set a matrix that applies a linear function:
 - ' Generate texture coordinates as linear functions
 - ' such that:
 - ' $u = U_x * x + U_y * y + U_z * z + U_w$
 - ' $v = V_x * x + V_y * y + V_z * z + V_w$
 - ' The matrix M for this case is:
 - ' $\begin{bmatrix} U_x & U_y & U_z & U_w \\ V_x & V_y & V_z & V_w \end{bmatrix}$

```
' Uz Vz 0 0  
' Uw Vw 0 0
```

```
SetTransform(D3DTRANSFORMSTATE_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 3 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 3-element projected texture coordinates.

```
' Two output coordinates will be used.  
SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,  
D3DTTF_PROJECTED | D3DTTFF_COUNT3)
```
-

Texture Surface Objects

This section presents information on creating and rendering with texture interface pointers in the following topics:

- Obtaining a Texture Surface Object
- Rendering with Texture Surfaces

Obtaining a Texture Surface Object

[C++]

Textures are simply DirectDraw surface objects, and therefore they expose the **IDirectDrawSurface7** interface. The first step in obtaining a pointer to a texture surface interface is to create a DirectDraw surface with the DDSCAPS_TEXTURE capability set. See Creating Surfaces and **DDSCAPS2**.

Once the surface is created, your application has a pointer to the texture surface's **IDirectDrawSurface7** interface. You can use the methods of the surface interface to manipulate the data it contains the same as you would with any other surface. The **IDirectDrawSurface7** of a texture serves as its identifier; all of the texture-related methods in the **IDirect3DDevice7** interface accept pointers to this interface.

Note

Use the **IDirect3DDevice7::Load** method to load images into your textures. Loading textures into video memory by calling this method is preferred over blit operations.

[\[Visual Basic\]](#)

Textures are simply DirectDraw surface objects, represented by the **DirectDrawSurface7** class. The first step in obtaining a texture is to create a DirectDraw surface with the DDSCAPS_TEXTURE capability set. See Creating Surfaces and **DDSCAPS2**.

Once the surface is created, your application has a reference to the surface class object. You can use the methods of **DirectDrawSurface7** to manipulate the data it contains the same as you would with any other surface. Textures are identified by the reference to the **DirectDrawSurface7** class that your application holds. All of the texture-related methods in the **Direct3DDevice7** class accept this class reference as a parameter.

Note

Use the **Direct3DDevice7.Load** method to load images into your textures. Loading textures into video memory by calling this method is preferred over blit operations.

Rendering with Texture Surfaces

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 **IDirect3DDevice7::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 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 **IDirect3DDevice7::SetTexture** method. Direct3D will blend 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 **IDirect3DDevice7::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 to enable wrapping in the u or v directions.

Your application can also set the texture perspective and texture filtering states. See Texture Perspective State, Texture Filtering, and Texture Filtering State.

[\[Visual Basic\]](#)

From 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 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 will blend 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 Perspective State, Texture Filtering, and Texture Filtering State.

Texture Filtering

When 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 developers with three types of texture filtering—linear filtering, anisotropic filtering, and mipmap filtering. If you select no texture filtering, Direct3D utilizes 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. On the other hand, 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 **IDirect3DDevice7::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 either **D3DTSS_MAGFILTER**, **D3DTSS_MINFILTER**, or **D3DTSS_MIPFILTER** as the value of the second parameter. Set the third parameter to a member of the **D3DTEXTUREMAGFILTER**, **D3DTEXTUREMINFILTER**, or **D3DTEXTUREMIPFILTER** enumerated types respectively.

[Visual Basic]

Visual Basic applications should set the current texture filtering method by calling the **Direct3DDevice7.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 either **D3DTSS_MAGFILTER**, **D3DTSS_MINFILTER**, or **D3DTSS_MIPFILTER** as the value of the second parameter. Set the third parameter to a member of the **CONST_D3DTEXTUREMAGFILTER**, **CONST_D3DTEXTUREMINFILTER**, or **CONST_D3DTEXTUREMIPFILTER** enumerated types respectively.

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 **IDirect3DDevice7::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 multi-texture 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. 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 will need to 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 **IDirect3DDevice7::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, and set the second parameter to **D3DTSS_MAGFILTER**, **D3DTSS_MINFILTER**, or **D3DTSS_MIPFILTER**. Pass a member of **D3DTEXTUREMAGFILTER**, **D3DTEXTUREMINFILTER**, or **D3DTEXTUREMIPFILTER** enumerated types as the value in the next parameter to set the magnification, minification, or mipmapping filter.

For more information, see Texture Filtering State.

[Visual Basic]

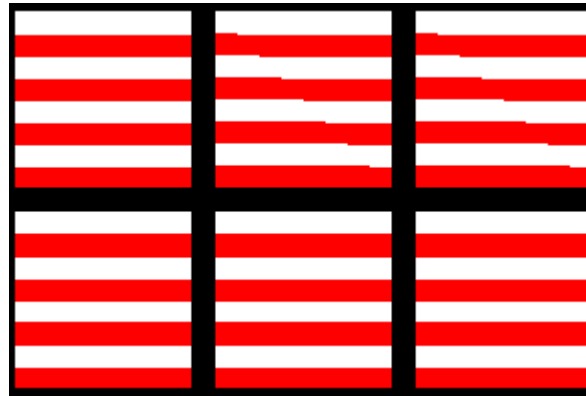
Your Visual Basic application can select nearest point sampling by calling the **Direct3DDevice7.SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method, and set the second parameter to **D3DTSS_MAGFILTER**, **D3DTSS_MINFILTER**, or **D3DTSS_MIPFILTER**. Pass a member of the **CONST_D3DTEXTUREMAGFILTER**, **CONST_D3DTEXTUREMINFILTER**, or **CONST_D3DTEXTUREMIPFILTER** enumerations in the next 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.)

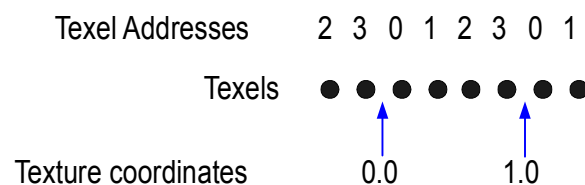
This effect can be seen when mapping a very small texture onto a very large polygon: an operation often called "magnification." For example, when using a texture that looks like a checkerboard, nearest point sampling results in a larger checkerboard that shows distinct edges. By contrast, linear texture filtering results in an image where the checkerboard colors vary smoothly across the polygon.

In most cases, applications can 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 screen capture shows what these artifacts can look like:



Notice that the two squares in the top-right of the group appear different than their neighbors. To avoid graphic artifacts like these, you should start by understanding 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 D3DTEXTADDRESS_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 will be the 0th 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 0th 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 will result in pixels for which the texture map is sampled at the boundary between texels. Any inaccuracies in the computation of texture coordinates, however small, will result 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 somewhat because the errors are accumulated evenly during iteration.

The Direct3D reference rasterizer uses a direct-evaluation approach for computing texture indices 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 occurring at the boundaries can be more noticeable since the reference rasterizer does not perform this operation with perfect accuracy (which is, again, difficult and very expensive with floating point texture coordinates).

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

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. Just as in nearest point sampling, it then finds the texel whose integer address is closest to the computed address. In addition, the Direct3D rendering module will compute 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 **IDirect3DDevice7::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** as the value of the second parameter if you are setting the magnification filter. Pass **D3DTEXTUREMINFILTER** as the value of the second parameter if you are setting the minification filter. Pass **D3DTEXTUREMIPFILTER** as the value of the second parameter if you are setting the mipmapping filter. Set the third parameter to **D3DTFG_LINEAR** if you are setting the magnification filter, **D3DTFN_LINEAR** if you are setting the minification filter, or **D3DTFP_LINEAR** if you are setting the mipmap filter. For more information, see [Texture Filtering State](#).

[Visual Basic]

Select bilinear texture filtering by invoking the **Direct3DDevice7.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 a member of the **CONST_D3DTEXTUREMAGFILTER**, **CONST_D3DTEXTUREMINFILTER**, or **CONST_D3DTEXTUREMIPFILTER** enumerations in the second parameter to set the magnification, minification, or mipmapping filter. Set the third parameter to **D3DTFG_LINEAR** if you are setting the magnification filter, **D3DTFN_LINEAR** if you are setting the minification filter,

or `D3DFTP_LINEAR` if you are setting the mipmap filter. 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 into texels, its shape is distorted. Direct3D measures the anisotropy of a pixel as the elongation (length divided by width) of a screen pixel that is inverse-mapped into texture space.

[C++]

Anisotropic texture filtering can be used in conjunction with linear texture filtering or mipmap texture filtering to improve rendering results. Your application enables anisotropic texture filtering by calling the **IDirect3DDevice7::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 as the value of the second parameter if you are setting the magnification filter. Pass **D3DTEXTUREMINFILTER** as the value of the second parameter if you are setting the minification filter. Set the third parameter to `D3DTFG_ANISOTROPIC` if you are setting the magnification filter, or `D3DTFN_ANISOTROPIC` if you are setting the minification filter. For more information, see [Texture Filtering State](#).

Your program must also set the degree of anisotropy to a value greater than one. Do this by calling the **IDirect3DDevice7::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.

Disable isotropic filtering by setting the degree of isotropy to one (any value larger than one enables it). Check the `D3DPRASERCAPS_ANISOTROPY` flag in the **D3DPRIMCAPS** structure to determine the possible range of values for the degree of anisotropy.

[Visual Basic]

Anisotropic texture filtering can be used in conjunction with linear texture filtering or mipmap texture filtering to improve rendering results. Your application enables anisotropic texture filtering by calling the **Direct3DDevice7.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 a member of the **CONST_D3DTEXTUREMAGFILTER**, **CONST_D3DTEXTUREMINFILTER**, or **CONST_D3DTEXTUREMIPFILTER** enumerations in the second parameter to set the magnification, minification, or mipmapping filter. Set the third parameter to `D3DTFG_ANISOTROPIC` if you are setting the magnification filter, or

D3DTFN_ANISOTROPIC if you are setting the minification filter. 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 **Direct3DDevice7.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.

Disable isotropic filtering by setting the degree of isotropy to one (any value larger than one enables it). Check the D3DPRASERCAPS_ANISOTROPY flag in the **D3DPRIMCAPS** 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 for rendering 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 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 viewer. Lower-resolution images are used as the object moves farther away. Mipmapping improves the quality of rendered textures at the expense of using more memory.

Direct3D represents mipmaps as a chain of attached surfaces. The highest resolution texture is at the head of the chain and has, as an attachment, the next level of the mipmap. That level has, in turn, an attachment that is the next level in the mipmap, and so on down to the lowest resolution level of the mipmap.

The following set of illustrations shows an example. The set of bitmap textures represents a sign on the side of 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 a power of 2 smaller in height and width. 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.

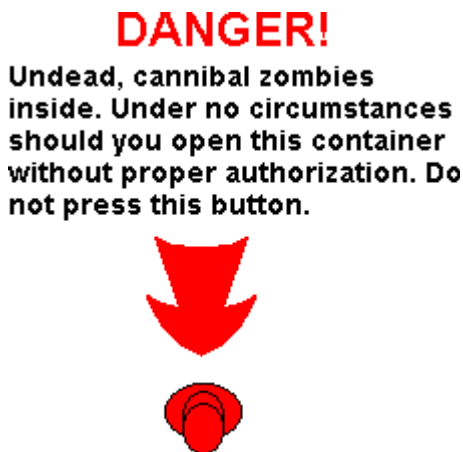
This sign would have a maximum distance from which it is visible. If the player begins far away from the sign, the game would display the smallest texture in the mipmap chain, which in this case the 64x64 texture.



As the player moves the point of view closer to the sign, progressively higher-resolution textures in the mipmap chain are used.



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 effects for textures. Rather than render a single texture to many resolutions, it is faster to use multiple textures at varying resolutions.

Direct3D is able to assess which texture in a mipmap set is the closest resolution to the desired output and 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 of the mipmaps and blend their color values together.

If you want your application to use mipmaps, it 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 program 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 `D3DFTP_POINT` enumerated value to select this. Direct3D can produce better filtering results if your application uses the `D3DFTP_LINEAR` enumerated value. This will select the nearest mipmap, then compute a weighted average of the texels surrounding the location in the texture to which the current pixel maps.

Creating a Set of Mipmaps

[C++]

To create a mipmap chain, specify the `DDSCAPS_MIPMAP` and `DDSCAPS_COMPLEX` flags in the **DDSURFACEDESC2** structure passed to the **IDirectDraw7::CreateSurface** method. Because all mipmaps are also textures, the `DDSCAPS_TEXTURE` flag must also be specified.

The following example demonstrates how your application can use the **IDirectDraw7::CreateSurface** method to build a chain of five mipmap levels of sizes 256×256 , 128×128 , 64×64 , 32×32 , and 16×16 :

```
// This code fragment assumes that the variable lpDD is a
// valid pointer to a DirectDraw interface.
```

```
DDSURFACEDESC2    ddsd;
LPDIRECTDRAW7 lpDDMipMap;
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_MIPMAPCOUNT;
ddsd.dwMipMapCount = 5;
ddsd.ddsCaps.dwCaps = DDSCAPS_TEXTURE |
    DDSCAPS_MIPMAP | DDSCAPS_COMPLEX;
ddsd.dwWidth = 256UL;
ddsd.dwHeight = 256UL;

ddres = lpDD->CreateSurface(&ddsd, &lpDDMipMap);
if (FAILED(ddres))
```

```
.
.
.
```

You can omit the number of mipmap levels, in which case the **IDirectDraw7::CreateSurface** method will create a chain of surfaces, each a power of two smaller than the previous one, down to the smallest possible size. It is also possible to omit the width and height, in which case **IDirectDraw7::CreateSurface** will create the number of levels you specify, with a minimum level size of 1×1 .

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 2×1 . If you explicitly specify dimensions in the **dwWidth** and **dwHeight** members, you should be aware of some restrictions. Namely, you cannot request a number of mipmap levels in **dwMipMapCount** that would cause either the width or height of any mipmap in the chain to be smaller than 1. Take the very simple case of a 4×2 top-level mipmap surface: the maximum value allowed for **dwMipMapCount** here is 2: the top-level dimensions are 4×2 , and the dimensions for the second level 2×1 . A value larger than 2 in **dwMipMapCount** would result in a fractional value in the height of the second-level mipmap, and is therefore disallowed.

[Visual Basic]

To create a mipmap chain, specify the **DDSCAPS_MIPMAP** and **DDSCAPS_COMPLEX** flags in the **DDSURFACEDESC2** type passed to the **DirectDraw7.CreateSurface** method. Because all mipmaps are also textures, the **DDSCAPS_TEXTURE** flag must also be specified.

The following example demonstrates how your application can use the **DirectDraw7.CreateSurface** method to build a chain of five mipmap levels of sizes 256×256 , 128×128 , 64×64 , 32×32 , and 16×16 :

```
' This code fragment assumes that the variable DD contains
' a valid reference to a DirectDraw7 object.
On Local Error Resume Next
Dim ddsd As DDSURFACEDESC2
Dim DDMipMap As DirectDrawSurface7

ddsd.lFlags = DDSD_CAPS Or DDSD_MIPMAPCOUNT
ddsd.lMipMapCount = 5
ddsd.ddsCaps.lCaps = DDSCAPS_TEXTURE Or DDSCAPS_MIPMAP Or
DDSCAPS_COMPLEX
ddsd.lWidth = 256
ddsd.lHeight = 256

Set DDMipMap = DD.CreateSurface(ddsd)
If Err.Number <> DD_OK Then
    ' Code to handle error goes here
End If
```

You can omit the number of mipmap levels, in which case the **DirectDraw7.CreateSurface** method will create a chain of surfaces, each a power of two smaller than the previous one, down to the smallest possible size. It is also possible to omit the width and height, in which case **DirectDraw7.CreateSurface** will create the number of levels you specify, with a minimum level size of 1×1 .

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 2×1. If you explicitly specify dimensions in the **IWidth** and **IHeight** members, you should be aware of some restrictions. Namely, you cannot request a number of mipmap levels in **IMipMapCount** that would cause either the width or height of any mipmap in the chain to be smaller than 1. Take the very simple case of a 4×2 top-level mipmap surface: the maximum value allowed for **IMipMapCount** here is 2: the top-level dimensions are 4×2, and the dimensions for the second level 2×1. A value larger than 2 in **IMipMapCount** would result in a fractional value in the height of the second-level mipmap, and is therefore disallowed.

Selecting and Displaying a Mipmap

[C++]

Call the **IDirect3DDevice7::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 **D3DTEXTUREMIPFILTER** enumerated type to the D3DTSS_MIPFILTER texture stage state. Direct3D will then automatically perform mipmap texture filtering.

Your application can also manually traverse a chain of mipmap surfaces by using the **IDirectDrawSurface7::GetAttachedSurface** method and specifying the DDSCAPS_MIPMAP and DDSCAPS_TEXTURE flags in the **DDSCAPS2** structure. The following example traverses a mipmap chain from highest to lowest resolutions:

```
LPDIRECTDRAW_SURFACE7 lpDDLevel, lpDDNextLevel;
DDSCAPS2 ddsCaps;
HRESULT ddres;

lpDDLevel = lpDDMipMap;
lpDDLevel->AddRef();
ddsCaps.dwCaps = DDSCAPS_TEXTURE | DDSCAPS_MIPMAP;
ddres = DD_OK;
while (ddres == DD_OK)
{
    // Process this level.
    .
    .
    .
    ddres = lpDDLevel->GetAttachedSurface(
&ddsCaps, &lpDDNextLevel);
    lpDDLevel->Release();
    lpDDLevel = lpDDNextLevel;
}
```

```

if ((ddres != DD_OK) && (ddres != DDERR_NOTFOUND))
{
    // Code to handle the error goes here
}
.
.
.

```

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.

Direct3D explicitly stores the number of levels in a mipmap chain. When an application obtains the surface description of a mipmap (by calling the **IDirectDrawSurface7::Lock** or **IDirectDrawSurface7::GetSurfaceDesc** method), the **dwMipMapCount** member of the **DDSURFACEDESC2** structure contains the number of levels in the mipmap, including the top level. For levels other than the top level in the mipmap, the **dwMipMapCount** member specifies the number of levels from that mipmap to the smallest mipmap in the chain.

[\[Visual Basic\]](#)

Call the **Direct3DDevice7.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_D3DTEXTUREMIPFILTER** enumerated type to the **D3DTSS_MIPFILTER** texture stage state. Direct3D will then automatically perform mipmap texture filtering.

Your application can also manually traverse a chain of mipmap surfaces by using the **DirectDrawSurface7.GetAttachedSurface** method and specifying the **DDSCAPS_MIPMAP** and **DDSCAPS_TEXTURE** flags in the **DDSCAPS2** type. The following example traverses a mipmap chain from highest to lowest resolutions:

On Local Error Resume Next

```

Dim DDLevel As DirectDrawSurface7, _
    DDNextLevel As DirectDrawSurface7

```

```

Dim ddsCaps As DDSCAPS2

```

```

DDLevel = DDMipMap
ddsCaps.ICaps = DDSCAPS_TEXTURE Or DDSCAPS_MIPMAP
While Err.Number = DD_OK
    ' Process this level.
    .
    .
    .

```

```
Set DDNextLevel = DDLevel.GetAttachedSurface(ddsCaps)
DDLevel = DDNextLevel
Wend
If ((Err.Number <> DD_OK) And (Err.Number <> DDERR_NOTFOUND)) Then
    ' Code to handle the error goes here
End If
.
.
.
```

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.

Direct3D explicitly stores the number of levels in a mipmap chain. When an application obtains the surface description of a mipmap (by calling the **DirectDrawSurface7.Lock** or **DirectDrawSurface7.GetSurfaceDesc** method), the **IMipMapCount** member of the **DDSURFACEDESC2** type contains the number of levels in the mipmap, including the top level. For levels other than the top level in the mipmap, the **IMipMapCount** member specifies the number of levels from that mipmap to the smallest mipmap in the chain.

Texture Wrapping

This section presents information on wrapping textures around 3-D primitives. The discussion is divided into the following topics:

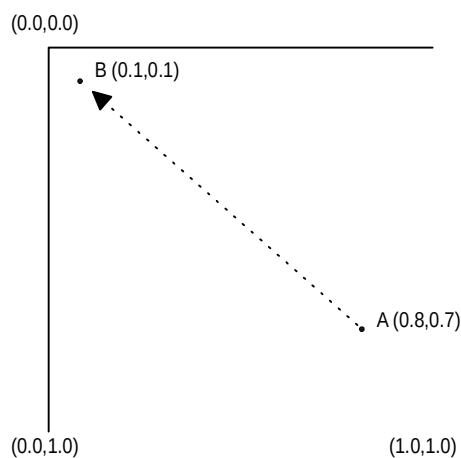
- What Is Texture Wrapping?
- Using Texture Wrapping

Note

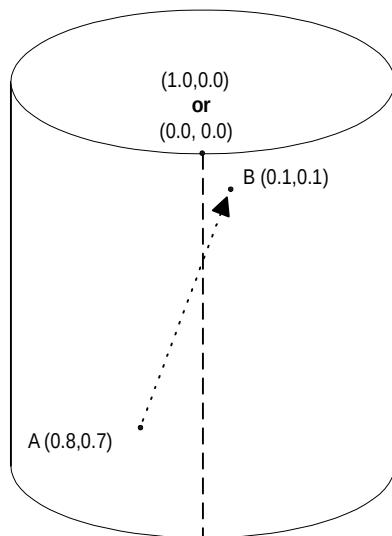
Texture wrapping should not be confused with the similarly named texture addressing modes. For more information, see [Texture Addressing Modes and Texture Wrapping](#) and [Texture Addressing Modes](#).

What Is Texture Wrapping?

Texture wrapping, in short, changes the basic way in which Direct3D rasterizes textured polygons using the texture coordinates specified for each vertex. (Don't confuse texture wrapping with the "wrap" texture addressing mode. For more information, see [Texture Addressing Modes and Texture Wrapping](#).) 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 would look like:



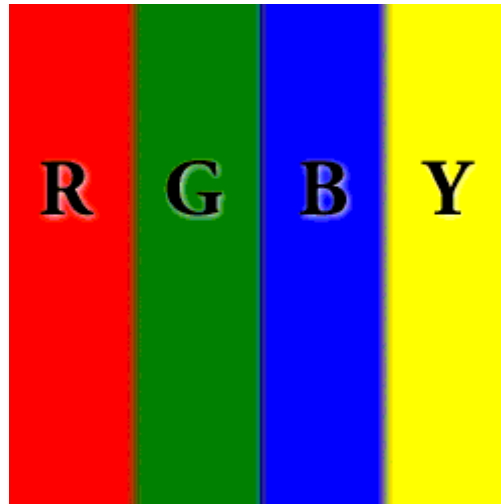
Note that the shortest distance between A and B in the preceding illustration runs roughly through the middle of the texture. Enabling u- or v-texture coordinate wrapping changes how Direct3D perceives the shortest route between texture coordinates in the u- and v-directions. By definition, texture wrapping 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, take the following illustration:



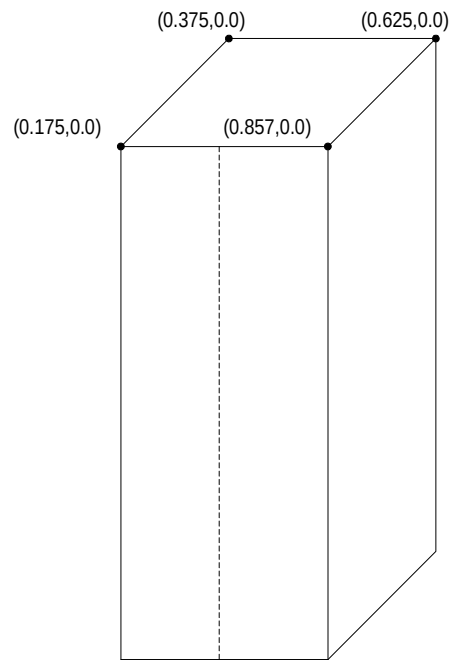
The preceding diagram shows how wrapping in the u- direction affects the way the system interpolates texture coordinates. Using the same points we used in the example for "normal", or non-wrapped, textures, you can see that the shortest route between points A and B is no longer across the middle of the texture; it's now across the border where 0.0 and 1.0 exist together. Wrapping in the v-direction is similar,

only it "wraps" the texture around a cylinder that is lying on its side. Wrapping in both the u- and v-directions is a little more complex. In this situation, you might envision the texture as a torus, or doughnut.

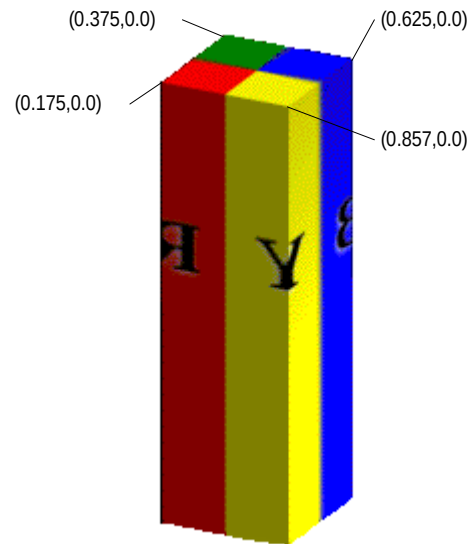
The most common practical application for texture wrapping is to perform environment mapping. Usually, an object textured with an environment map appears very reflective, showing a mirrored image of the object's surroundings in the scene. For the sake of this discussion, picture a room with four walls, 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:



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 pretty simple—making it look as though it's reflecting the letters and colors as they appear on the walls isn't 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 would look something like the following:



If texture wrapping wasn't enabled, the rasterizer would not interpolate in the direction needed to generate a believable, reflected, image. Rather, the area at the front of the pillar would contain 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 effect would be ruined.

Using Texture Wrapping

[C++]

In a C++ application, the process of enabling texture wrapping differs across versions the Direct3D device interfaces. If your application uses the **IDirect3DDevice7** interface, you enable texture wrapping for texture coordinate sets used by vertices, not for the texture stages themselves. In this case, call the

IDirect3DDevice7::SetRenderState method to enable texture wrapping, passing one of the D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7 enumerated values as the first parameter to identify which texture coordinate set will receive wrapping. 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 particular set of texture coordinates, set the value for the corresponding render state to 0.

If your application uses the legacy **IDirect3DDevice2** interface, you still enable texture wrapping by calling the **SetRenderState** method; however, these interface versions do not support D3DRENDERSTATE_WRAP0 through

D3DRENDERSTATE_WRAP7. Instead, use the D3DRENDERSTATE_WRAPU or D3DRENDERSTATE_WRAPV value in the first parameter. Specify TRUE as in the second parameter to enable wrapping, or FALSE to disable it.

Note

The **IDirect3DDevice7** interface does recognize the legacy D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV render states, even though they were superseded by D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7. These older render states, when passed to the **IDirect3DDevice7** version of **SetRenderState**, affect u- and v-texture wrapping for the first set of texture coordinates.

[Visual Basic]

In Visual Basic, you enable texture wrapping for texture coordinate sets used by vertices. Call the **Direct3DDevice7.SetRenderState** method to enable texture wrapping, passing one of the D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7 enumerated values as the first parameter to identify which texture coordinate set will receive wrapping. Use the D3DWRAPCOORD_0 through D3DWRAPCOORD_3 flags (from the **CONST_D3D** enumeration) in the second parameter to enable texture wrapping in the corresponding direction, or use combinations 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 particular set of texture coordinates, set the value for the corresponding render state to 0.

Texture Blending

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++]

If you want your C++ application to use texture blending, the program should first check to see if the user's hardware supports it. The relevant information is found in the **dwTextureCaps** member of the **D3DPRIMCAPS** structure. For details on how to query the user's hardware for texture blending capabilities, see **IDirect3DDevice7::GetCaps** and **D3DDEVICEDESC7**.

[Visual Basic]

If you want your Visual Basic application to use texture blending, it should first check to see if the user's hardware supports it. The relevant information is found in the **ITextureCaps** member of the **D3DPRIMCAPS** type. For details on how to query the user's hardware for texture blending capabilities, see **Direct3DDevice7.GetCaps** and **D3DDEVICEDESC7**.

Alpha Texture Blending

When 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 *Lighting and Materials*. If an application enables texture blending, Direct3D must then blend the texel colors of one or more textures with the primitive's current colors. Direct3D uses the following formula to determine the final color for each pixel in the primitive's image.

$$\textit{FinalColor} = \textit{TexelColor} \times \textit{SourceBlendFactor} + \textit{PixelColor} \times \textit{DestBlendFactor}$$

In the preceding formula, *FinalColor* is the pixel color that is output to the target rendering surface. *TexelColor* stands for the color of the texel that corresponds to the current pixel. For details on how Direct3D maps pixels to texels, see *Texture Filtering*. *SourceBlendFactor* is a calculated value that Direct3D uses to determine the percentage of the texel color to apply to the final color. *PixelColor* is the color of the current pixel in the primitive's image. *DestBlendFactor* represents the percentage of the current pixel's color that will be used in the final color. The values of *SourceBlendFactor* and *DestBlendFactor* range from 0.0 or 1.0 inclusive.

As you can see from the preceding formula, a texture is not rendered as transparent at all if the *SourceBlendFactor* is 1.0 and the *DestBlendFactor* is 0.0. It is completely transparent if the *SourceBlendFactor* is 0.0 and the *DestBlendFactor* is 1.0. If an application sets these factors to any other values, the resulting texture will be blended with some degree of transparency.

Every texel in a texture has a red, a green, and a blue color value. By default, Direct3D uses the alpha values of texels as the *SourceBlendFactor*. Therefore, applications can control the transparency of textures by setting the alpha values in their textures.

[C++]

An application developed with C++ controls the blending factors with the **D3DRENDERSTATE_SRCBLEND** and **D3DRENDERSTATE_DESTBLEND** render states. Invoke the **IDirect3DDevice7::SetRenderState** method and pass either **D3DRENDERSTATE_SRCBLEND** or **D3DRENDERSTATE_DESTBLEND** 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 can control the blending factors by using the `D3DRENDERSTATE_SRCBLEND` and `D3DRENDERSTATE_DESTBLEND` render states. Invoke the **Direct3DDevice7.SetRenderState** method and pass either render state value in the first parameter. The second parameter must be a member of the **CONST_D3DBLEND** enumeration.

Multipass Texture Blending

Direct3D applications can achieve numerous special effects by applying various textures onto 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 onto primitives in a single pass. For details, see *Multiple Texture Blending*.

If your user's hardware does not support multiple texture blending, your application can use multipass texture blending to achieve the same visual effects. However, it will not be able to sustain the frame rates that are possible when using multiple texture blending.

[C++]

0 To perform multipass texture blending in a C++ application

1. Set a texture in texture stage 0 by calling the **IDirect3DDevice7::SetTexture** method.
 2. Select the desired color and alpha blending arguments and operations with the **IDirect3DDevice7::SetTextureStageState** method. (The default settings are well-suited for multi-pass texture blending.)
 3. Render the appropriate objects in the scene.
 4. Set the next texture in texture stage 0.
 5. Set the `D3DRENDERSTATE_SRCBLEND` and `D3DRENDERSTATE_DESTBLEND` render states to adjust the source and destination blending factors as needed. The system will blend 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]

0 To perform multipass texture blending in a Visual Basic application

1. Set a texture in texture stage 0 by calling the **Direct3DDevice7.SetTexture** method.
 2. Select the desired color and alpha blending arguments and operations with the **Direct3DDevice7.SetTextureStageState** method. (The default settings are well-suited for multi-pass texture blending.)
 3. Render the appropriate objects in the scene.
 4. Set the next texture in texture stage 0.
 5. Set the **D3DRENDERSTATE_SRCBLEND** and **D3DRENDERSTATE_DESTBLEND** render states to adjust the source and destination blending factors as needed. The system will blend 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

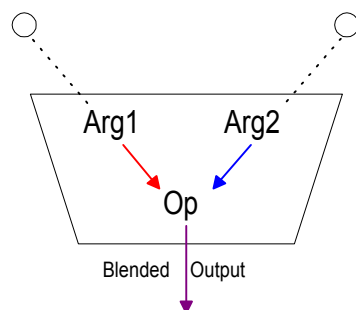
Direct3D can blend as many as eight textures onto primitives in a single pass. The use of multiple texture blending can profoundly increase the frame rate of Direct3D applications. Applications employ multiple texture blending to apply textures, shadows, specular lighting, diffuse lighting, and other special effects in a single pass.

To blend multiple textures, applications assign textures into the set of current textures, and then create blending stages. The topics in the following list present 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

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 the result on for further processing or for rasterization. You could visualize a texture stage as shown in the following figure.

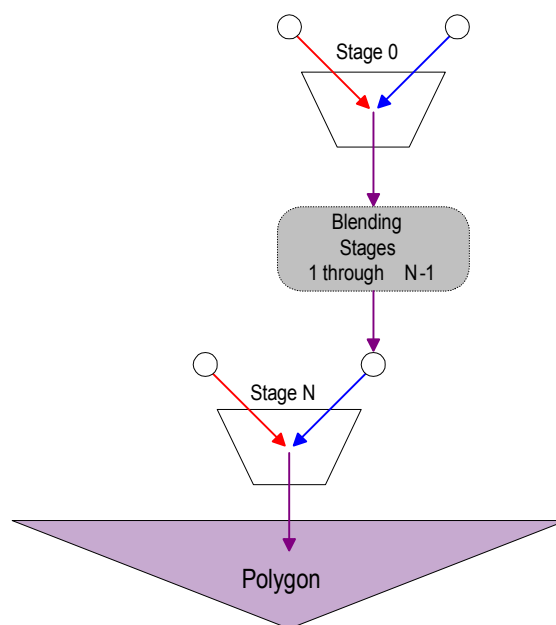


As the preceding illustration shows, texture stages blend two arguments by using a specified operator. Common operations include simple modulation or addition of the color or alpha components of the arguments, but more than two dozen operations are currently supported. The arguments for a stage can be 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, and then from that stage to the next, and so on. The idea that results flow 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 currently 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 order of index.

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, leaving the remaining stages alone. 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, 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 given texture stage is used by calling **IDirect3DDevice7::SetTextureStageState**. You can set separate operations for the color and alpha channels independently, and each operation uses two arguments. Specify color channel operations by using the D3DTSS_COLOROP stage state, and D3DTSS_ALPHAOP for alpha operations—both 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.

[\[Visual Basic\]](#)

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 given texture stage is used by calling **Direct3DDevice7.SetTextureStageState**. You can set separate operations for the color and alpha channels independently, and each operation uses two arguments. Specify color channel operations by using the D3DTSS_COLOROP stage state, and D3DTSS_ALPHAOP for alpha operations—both 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.

Assigning the Current Textures

Direct3D maintains a list of up to eight current textures. It blends these textures onto all of the primitive it renders. Only textures created as texture interface pointers can be used in the set of current textures.

[C++]

Applications call the **IDirect3DDevice7::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 fragment demonstrates how a texture can be assigned into the set of current textures:

```
// This code fragment assumes that the variable lpd3dDev is a valid
// pointer to an IDirect3DDevice7 interface and lpd3dTexture is a valid
// pointer to an IDirectDrawSurface7 interface.

// Set the third texture.
lpd3dDev->SetTexture(2, lpd3dTexture);
```

[Visual Basic]

Applications written in Visual Basic call **Direct3DDevice7.SetTexture** 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.

This Visual Basic code fragment demonstrates how a texture can be assigned into the set of current textures:

```
' This code fragment assumes that the variable d3dDev is a valid
' reference to a Direct3DDevice7 object and d3dTexture is a valid
' reference to a DirectDrawSurface7 object.

' Set the third texture.
Call d3dDev.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, together with their arguments, that define how textures are blended. When making a blending stage, C++ applications invoke the **IDirect3DDevice7::SetTextureStageState** method. The first call specifies

the operation that will be performed. Two additional invocations define the arguments to which the operation will be applied. The following code fragment illustrates the creation of a blending stage:

```
// This example assumes that lpD3DDev is a valid pointer to an
// IDirect3DDevice7 interface.

// Set the operation for the 1st texture.
lpD3DDev->SetTextureStageState(0,D3DTSS_COLOROP,D3DTOP_ADD);

// Set arg1 for the texture operation.
lpD3DDev->SetTextureStageState(0, // First texture
    D3DTSS_COLORARG1, // Set color arg 1
    D3DTA_TEXTURE); // Color arg 1 value

// Set arg2 for the texture operation.
lpD3DDev->SetTextureStageState(0, // First texture
    D3DTSS_COLORARG2, // Set color arg 2
    D3DTA_DIFFUSE); // Color arg 2 value
```

Texel data in textures contain color and alpha values. Programs 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 Direct3D API, the following macros can also be inserted into your program to abbreviate the code required for creating texture blending stage.

```
#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, together with their arguments, that define how textures are blended. When making a blending stage, Visual Basic applications invoke the **Direct3DDevice7.SetTextureStageState** method. The first call specifies the operation that will be performed. Two additional invocations define the arguments to which the operation will be applied. The following code fragment illustrates the creation of a blending stage:

```
' This example assumes that d3dDev is a valid reference to a
' Direct3DDevice7 object.
```

' Set the color operation for the 1st texture.

Call `d3dDev.SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_ADD)`

' Set arg1 to the texture color.

Call `d3dDev.SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE)`

' Set arg2 to the iterated diffuse color.

Call `d3dDev.SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE)`

Texel data in textures contain color and alpha values. Programs 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. Direct3D supports multipass and multiple texture blending. These capabilities enable your application to render scenes whose appearance is much more realistic 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 contain 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 program 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 both the light map and the base texture in one pass. If your user's hardware provides for multiple texture blending, your application should take advantage of it when performing light mapping. It will significantly improve 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, it can 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). 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 program uses the texture filtering capabilities of 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 fragment illustrates how an application can use a texture as a monochrome light map:

```
// This example assumes that lpD3DDev 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.
lpD3DDev->SetTexture(0,lpTexLightMap);

// Set the color operation.
lpD3DDev->SetTextureStageState(0,D3DTSS_COLOROP,
    D3DTOP_SELECTARG1);

// Set argument 1 to the color operation.
lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG1,
    D3DTA_TEXTURE | D3DTA_ALPHAREPLICATE);
```

[Visual Basic]

The following Visual Basic code fragment illustrates how an application can use a texture as a monochrome light map:

```
' This example assumes that d3dDev is a valid reference to a
' Direct3DDevice7 object and that texLightMap is a valid reference
' to a DirectDrawSurface7 texture that contains monochrome light map data.

' Set the light map texture as the current texture.
Call d3dDev.SetTexture(0, texLightMap)
```

```
' Set the color operation.  
Call d3dDev.SetTextureStageState(0, D3DTSS_COLOROP, _  
D3DTOP_SELECTARG1)  
  
' Set argument 1 to the color operation.  
Call d3dDev.SetTextureStageState(0, D3DTSS_COLORARG1, _  
D3DTA_TEXTURE Or D3DTA_ALPHAREPLICATE)
```

Since 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 fragment demonstrates light mapping with RGB color data:

```
// This example assumes that lpD3DDev is a valid pointer to an  
// IDirect3DDevice7 interface and that lpTexLightMap is a valid  
// pointer to a texture that contains RGB light map data.  
  
// Set the light map texture as the 1st texture.  
lpD3DDev->SetTexture(0, lpTexLightMap);  
  
lpD3DDev->SetTextureStageState(0,D3DTSS_COLOROP,  
D3DTOP_MODULATE);  
  
lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG1,  
D3DTA_TEXTURE);  
  
lpD3DDev->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.

[Visual Basic]

The following Visual Basic code demonstrates light mapping with RGB color data:

' This example assumes that D3DDev is a valid reference to a
' IDirect3DDevice7 object and that texLightMap is a valid reference
' to a IDirectDrawSurface7 texture surface that contains RGB light map data.

' Set the light map texture as the 1st texture.
Call d3dDev.SetTexture(0, texLightMap)

Call d3dDev.SetTextureStageState(0, D3DTSS_COLOROP, _
D3DTOP_MODULATE)

Call d3dDev.SetTextureStageState(0, D3DTSS_COLORARG1, _
D3DTA_TEXTURE)

Call d3dDev.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) will receive specular highlights. In some cases, the specular highlights produced by the lighting module will not be accurate. To produce a more appealing highlight, many 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 fragment illustrates this process in C++:

```
// This example assumes that lpD3DDev is a valid pointer to an  
// IDirect3DDevice7 interface.  
// lpTexture is a valid pointer to a texture.  
// lpSpecLightMap is a valid pointer to a texture that contains RGB  
// specular light map data.  
// lpLightMap is a valid pointer to a texture that contains RGB  
// light map data.
```

```
// Set the base texture.  
lpD3DDev->SetTexture(0, lpTexture );
```

```
// Set the base texture operation and args  
lpD3DDev->SetTextureStageState(0, D3DTSS_COLOROP,  
    D3DTOP_MODULATE );
```

```

lpD3DDevice->SetTextureStageState(0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
lpD3DDevice->SetTextureStageState(0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// Set the specular light map.
lpD3DDevice->SetTexture(1,lpTexSpecLightMap );

// Set the specular light map operation and args
lpD3DDevice->SetTextureStageState(1,D3DTSS_COLOROP,
    D3DTOP_ADD );
lpD3DDevice->SetTextureStageState(1,D3DTSS_COLORARG1, D3DTA_TEXTURE );
lpD3DDevice->SetTextureStageState(1,D3DTSS_COLORARG2, D3DTA_CURRENT );

// Set the RGB light map.
lpD3DDevice->SetTexture(2,lpTexLightMap);

// Set the RGB light map operation and args
lpD3DDevice->SetTextureStageState(2,D3DTSS_COLOROP, D3DTOP_MODULATE);
lpD3DDevice->SetTextureStageState(2,D3DTSS_COLORARG1, D3DTA_TEXTURE );
lpD3DDevice->SetTextureStageState(2,D3DTSS_COLORARG2, D3DTA_CURRENT );

```

[Visual Basic]

The following Visual Basic code fragment illustrates this process:

```

' This example assumes that d3dDev is a valid reference to an
' Direct3DDevice7 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 d3dDev.SetTexture(0, texBaseTexture)

' Set the base texture operation and args
Call d3dDev.SetTextureStageState(0, D3DTSS_COLOROP, _
    D3DTOP_MODULATE)
Call d3dDev.SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDev.SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE)

' Set the specular light map.
Call d3dDev.SetTexture(1, texSpecLightMap)

' Set the specular light map operation and args
Call d3dDev.SetTextureStageState(1, D3DTSS_COLOROP, _

```

```

D3DTOP_ADD)
Call d3dDev.SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDev.SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT)

' Set the RGB light map.
Call d3dDev.SetTexture(2, texLightMap)

' Set the RGB light map operation and args
Call d3dDev.SetTextureStageState(2, D3DTSS_COLOROP, D3DTOP_MODULATE)
Call d3dDev.SetTextureStageState(2, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDev.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 fragment:

```

// This example assumes that lpD3DDev is a valid pointer to an
// IDirect3DDevice7 interface.
// lpTexture is a valid pointer to a texture.
// lpTextureDiffuseLightMap is a valid pointer to a texture that contains
// RGB diffuse light map data.

// Set the base texture.
lpD3DDev->SetTexture(0,lpTexture );

// Set the base texture operation and args
lpD3DDev->SetTextureStageState(0,D3DTSS_COLOROP,
    D3DTOP_MODULATE );
lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// Set the diffuse light map.
lpD3DDev->SetTexture(1,lpTextureDiffuseLightMap );

// Set the blend stage.
lpD3DDev->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE );
lpD3DDev->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );

```

```
lpD3DDevice->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 code Visual Basic fragment:

```
' This example assumes that d3dDev 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 d3dDev.SetTexture(0, texBaseTexture)

' Set the base texture operation and args
Call d3dDev.SetTextureStageState(0, D3DTSS_COLOROP, _
    D3DTOP_MODULATE)
Call d3dDev.SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDev.SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE)

' Set the diffuse light map.
Call d3dDev.SetTexture(1, texDiffuseLightMap)

' Set the blend stage.
Call d3dDev.SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE)
Call d3dDev.SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call d3dDev.SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT)
```

Texture Compression

DirectDraw provides services to compress surfaces that will be used for texturing 3-D models. For information on creating and manipulating the data in a compressed texture surface, see Compressed Texture Surfaces in the DirectDraw documentation.

[C++]

Once your application creates a rendering device, it can determine if the device supports texturing from compressed texture surfaces by calling the **IDirect3DDevice7::EnumTextureFormats** method. The method calls the **D3DEnumPixelFormatsCallback** callback function that you provide for each pixel format that the device supports for texture maps. If any of the enumerated pixel formats use the DXT1, DXT2, DXT3, DXT4, or DXT5 four character codes (FOURCCs), the device can texture directly from a compressed texture surface that uses that format. If so, you can use compressed texture surfaces directly with

Direct3D by calling the **IDirect3DDevice7::SetTexture** method. If the device doesn't 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. The DirectDraw blit methods, **IDirectDrawSurface7::Blt** and **IDirectDrawSurface7::BltFast**, automatically perform conversion of compressed formats to standard RGBA formats. To minimize loss of information, try to pick a destination pixel format that closely matches the compressed format. For instance, ARGB:1555 would be a good destination format for DXT1, but ARGB:4444 would be a better choice for DXT3 since DXT3 contains four bits of alpha information.

[\[Visual Basic\]](#)

Once your application creates a rendering device, it can determine if the device supports texturing from compressed texture surfaces by calling the **Direct3DDevice7.GetTextureFormatsEnum** method. The method returns a reference to a **Direct3DEnumPixelFormats** class object, which exposes methods to retrieve information about the supported texture surface pixel formats. The **Direct3DEnumPixelFormats.GetItem** method fills a **DDPIXELFORMAT** type to describe a supported texture format. If any of the enumerated pixel formats use the DXT1, DXT2, DXT3, DXT4, or DXT5 four character codes (FOURCCs), the device can texture directly from a compressed texture surface that uses that format. If so, you can use compressed texture surfaces directly with Direct3D by calling the **Direct3DDevice7.SetTexture** method. If the device doesn't 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. The DirectDraw blit methods, **DirectDrawSurface7.Blt** and **DirectDrawSurface7.BltFast**, automatically perform conversion of compressed formats to standard RGBA formats. To minimize loss of information, try to pick a destination pixel format that closely matches the compressed format. For instance, ARGB:1555 would be a good destination format for DXT1, but ARGB:4444 would be a better choice for DXT3 since DXT3 contains four bits of alpha information.

Automatic Texture Management

Texture management, in short, 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. Like 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 currently needed for rendering, and which aren't.

- Determining which of the existing texture surfaces can be reloaded with another texture image, and which surfaces should be destroyed and replaced with new texture surfaces.

Historically, Direct3D applications were responsible for managing textures on their own. Direct3D Immediate Mode for DirectX 6.0 introduced system-supported texture management, where Direct3D efficiently and intelligently performs texture management, ensuring that textures are loaded for optimal performance. (Texture surfaces that Direct3D manages are casually referred to as "managed textures.")

The texture manager uses a least-recently-used (LRU) algorithm to determine which textures should be evicted, which tracks textures with a time-stamp that identifies when the texture was last used. 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 will be removed in favor of the other texture. However, if two textures have the same time-stamp, the texture that has a lower priority will be evicted first.

[C++]

You request automatic texture management for textures surfaces when you create them. To get a managed texture in a C++ application, simply create a texture surface that also includes the `DDSCAPS2_TEXTUREMANAGE` or `DDSCAPS2_D3DTEXTUREMANAGE` flags in the **dwCaps2** member of the associated **DDSCAPS2** structure. Note that you are not allowed to specify where you want the texture created—you can't use the `DDSCAPS_SYSTEMMEMORY` or `DDSCAPS_VIDEOMEMORY` flags when creating a managed texture. After creating the managed texture, you can call the **IDirect3DDevice7::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 **IDirectDrawSurface7::SetPriority** method for the texture surface.

[Visual Basic]

You request automatic texture management for textures surfaces when you create them. To get a managed texture from Visual Basic, create a texture surface that also includes the `DDSCAPS2_TEXTUREMANAGE` or `DDSCAPS2_D3DTEXTUREMANAGE` flags in the **ICaps2** member of the associated **DDSCAPS2** type. Note that you are not allowed to specify where you want the texture created—you can't use the `DDSCAPS_SYSTEMMEMORY` or `DDSCAPS_VIDEOMEMORY` flags when creating a managed texture. After creating the managed texture, you can call the **Direct3DDevice7.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 **DirectDrawSurface7.SetPriority** method for the texture surface.

Direct3D automatically downloads textures into video memory as needed. (The system might cache managed textures in local or non-local video memory, depending on the availability of non-local video memory or other factors. Where your managed textures are cached is not communicated to your application, nor is this knowledge required to take advantage of automatic texture management.) If your application uses more textures than can fit in video memory, Direct3D will evict older textures from video memory to make room for the new textures. If you use an evicted texture again, the system uses the original system memory texture surface to reload the texture in the video memory cache. Reloading the texture is a minor, but obviously necessary, performance hit to the application.

You can dynamically modify the original system memory copy of the texture by blitting or locking the texture surface. When the system detects a "dirty" surface—after a blit 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 an evicted texture.

[C++]

When entering a new level in a game, your application may need to flush all managed textures from video memory. You can explicitly request that all managed textures be evicted by calling the **IDirect3D7::EvictManagedTextures** method. When you call this method, Direct3D destroys any cached local and non-local 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 that all managed textures be evicted by calling the **Direct3D7.EvictManagedTextures** method. When you call this method, Direct3D destroys any cached local and non-local video memory textures, but leaves the original system memory copies untouched.

Hardware Considerations for Texturing

Current hardware does not necessarily implement all of the functionality that the Direct3D interface enables. Your program must test user's 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 program 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 will need to 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 **IDirect3DDevice7**. Specifically, there is little support for setting the D3DTSS_MIPFILTER texture stage state to D3DTFP_LINEAR. Your application can use multipass texture blending to achieve the same effects, or degrade to the D3DTFP_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 the **Direct3DDevice7** Visual Basic class. Specifically, there is little support for setting the D3DTSS_MIPFILTER texture stage state to D3DTFP_LINEAR. Your application can use multipass texture blending to achieve the same effects, or degrade to the D3DTFP_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
-

[\[C++\]](#)

The Direct3D Immediate Mode C++ tutorials provide additional information about using depth buffers. See Tutorial 2: Adding a Depth Buffer (C++) for details.

[\[Visual Basic\]](#)

The Direct3D Immediate Mode Visual Basic tutorials provide additional information about using depth buffers. See Tutorial 2: Adding a Depth Buffer (Visual Basic) for details.

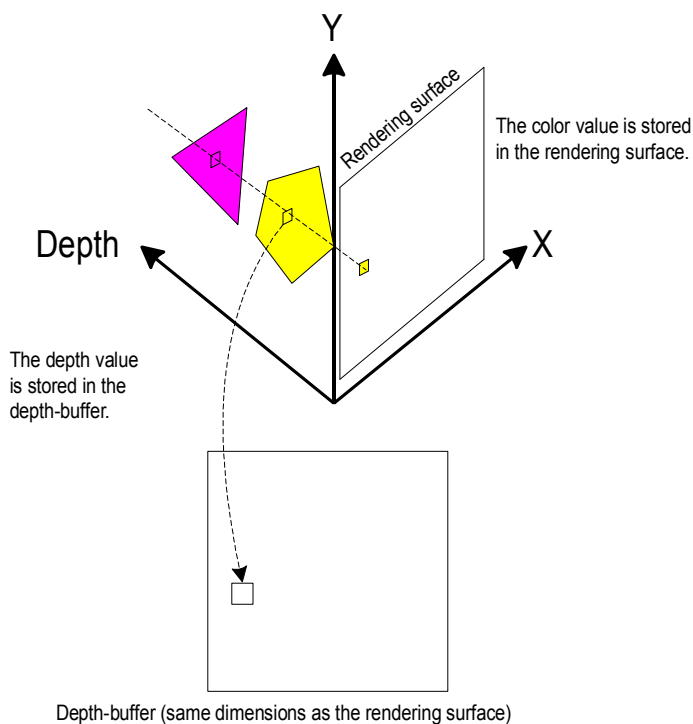
What Are Depth Buffers?

A depth buffer, often called a z-buffer or a w-buffer, is a DirectDraw surface that stores depth information to be used by Direct3D. When Direct3D renders a 3-D scene to a target surface, it can use the memory in an attached depth buffer surface as a workspace to determine how the pixels of rasterized polygons occlude one another. Direct3D uses an off-screen DirectDraw surface as the target to which final color values are written. The depth buffer surface that is attached to the render-target

surface is used to store depth information which 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 called a "w-buffer." Each type of depth buffer has its 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 already 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 Direct3D uses to determine which values will be placed in the depth buffer and subsequently the render-target surface. To do so, change the value for the `D3DRENDERSTATE_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 will be placed in the depth buffer and subsequently the render-target surface. To do so, change the value for the `D3DRENDERSTATE_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 do 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 first 2% of the depth range, and the distribution gets 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 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.

[\[C++\]](#)

The DirectDraw HEL can create depth buffers for use by Direct3D or other 3-D-rendering software. The HEL supports 16-bit depth buffers. The DirectDraw device driver for a 3-D accelerator can permit the creation of depth buffers in display memory by exposing the `DDSCAPS_ZBUFFER` flag. You can query for the supported depth buffer bit-depths by calling the **IDirect3D7::EnumZBufferFormats** method.

Whenever a z-buffer surface is created, your application should maintain a pointer to the z-buffer until it shuts the Direct3D system down. Your program should release the z-buffer surface just before it releases the rendering surface.

[Visual Basic]

The DirectDraw HEL can create depth buffers for use by Direct3D or other 3-D–rendering software. The HEL supports 16-bit depth buffers. The DirectDraw device driver for a 3-D accelerator can permit the creation of depth buffers in display memory by exposing the DDSCAPS_ZBUFFER flag. You can query for the supported depth buffer bit-depths by using the

Direct3D7.GetEnumZBufferFormats method. The **GetEnumZBufferFormats** method returns a reference to a **Direct3DEnumPixelFormat** class object that contains the supported pixel formats for depth buffers.

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
- Clearing Depth Buffers
- Changing Depth Buffer Write Access
- Changing Depth Buffer Comparison Functions
- Using Z-Bias

See Also

What Are Depth Buffers?, Tutorial 2: Adding a Depth Buffer (C++), Tutorial 2: Adding a Depth Buffer (Visual Basic)

Querying for Depth Buffer Support

As with any feature, don't assume that the driver your application uses supports depth buffering; you should always check the driver's capabilities. Although most drivers support z-based depth buffering, not all will be able to provide support to w-based depth buffering. (For general information about depth buffering, see What Are Depth Buffers?) Drivers don't fail if you attempt to enable an unsupported scheme, falling back on another depth buffering method instead, or sometimes disabling 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 the DirectDraw for the display device your application uses before you create a Direct3D device. If the DirectDraw object reports that it supports depth buffering, any hardware devices you create from this DirectDraw object will support z-buffering (but you don't yet know if the driver supports w-buffering).

[C++]

0 To query for general depth buffering support from C++

1. Call the **IDirectDraw7::GetCaps** method of the DirectDraw object for the display device your application uses, passing initialized **DDCAPS** structures as parameters. After the call, the **DDCAPS** structures contain information about hardware and emulation capabilities of DirectDraw.
 2. Examine the **ddsCaps** member of the structure you passed as the first parameter. If this member—a **DDSCAPS2** structure—includes the **DDSCAPS_ZBUFFER** flag, the driver supports depth buffering through z-buffers.
-

[Visual Basic]

0 To query for general depth buffering support in Visual Basic

1. Call the **DirectDraw7.GetCaps** method of the DirectDraw object for the display device your application uses, passing variables of type **DDCAPS** as parameters. After the call, the **DDCAPS** variables contain information about hardware and emulation capabilities of DirectDraw.
 2. Examine the **ddsCaps** member of the variable you passed as the first parameter. If this member—a **DDSCAPS2** type—includes the **DDSCAPS_ZBUFFER** flag, the driver supports depth buffering through z-buffers.
-

Once you know that the driver supports z-buffers, you can verify w-buffer support. Although z-buffers are supported for all software rasterizers, w-buffers are only supported by the reference rasterizer, which is hardly suited for use by real-world applications. No matter what type of device your application uses, you should verify support for w-buffers before you attempt to enable w-based depth buffering.

[C++]

0 To determine support for w-buffers in a C++ application

1. After creating your device (HAL or emulated), call the **IDirect3DDevice7::GetCaps** method, passing an initialized **D3DDEVICEDESC7** structure.

-
2. After the call, the **dpcTriCaps** and **dpcLineCaps** members (**D3DPRIMCAPS** structures) contain information about the driver's support for rendering primitives.
 3. If the **dwRasterCaps** member of these structures contains the **D3DPRASTERCAPS_WBUFFER** flag, then the driver supports w-based depth buffering for that primitive type.
-

[Visual Basic]

0 To determine support for w-buffers in Visual Basic

1. After creating your device (HAL or emulated), call the **Direct3DDevice7.GetCaps** method, passing a variable of type **D3DDEVICEDESC7** type as a parameter.
 2. After the call, the **dpcTriCaps** and **dpcLineCaps** members (**D3DPRIMCAPS** types) each contain information about the driver's support for rendering primitives.
 3. If the **IRasterCaps** members of **dpcTriCaps** and **dpcLineCaps** contain the **D3DPRASTERCAPS_WBUFFER** flag, the driver supports w-based depth buffering for that primitive type.
-

Creating a Depth Buffer

You usually create a depth buffer during your application's startup sequence, before creating a Direct3D device object for rendering. Use the following steps to create and attach a depth buffer surface to the render-target surface:

[C++]

0 To create a depth buffer in a C++ application

1. Call the **IDirect3D7::EnumZBufferFormats** method to determine the depth-buffer pixel formats that the device supports.
2. Prepare a **DDSURFACEDESC2** structure that describes a DirectDraw surface that matches the render-target surface's dimensions, includes the **DDSCAPS_ZBUFFER** capability flag, and uses a supported depth-buffer pixel format (retrieved in Step 1).
3. Create the surface in video memory or system memory, depending on what type of rendering device your application will use. (See note.)
4. Attach the depth buffer surface to the rendering surface using the **IDirectDrawSurface7::AddAttachedSurface** method.

After creating the depth buffer surface and attaching it to the render-target surface, call the **IDirect3D7::CreateDevice** method to create a rendering device that uses the render-target surface and its depth buffer.

Create the depth buffer in video memory—with the DDSCAPS_VIDEOMEMORY surface capability—when your application uses a hardware driver (HAL device), and in system memory—the DDSCAPS_SYSTEMMEMORY surface capability—when using software emulation drivers (the MMX or RGB devices). Failing to create a depth buffer in the appropriate type of memory will cause the **CreateDevice** method to fail.

Note

Some popular hardware devices require that the render target and depth buffer surfaces use the same bit depth. On such hardware, if your application uses a 16-bit render-target surface, the attached depth buffer must also be 16-bits. For a 32-bit render-target surface, the depth buffer must be 32-bits, of which 8-bits can be used for stencil buffering (if needed).

If the hardware upon which your application is running has this requirement, and your application fails to meet it, any attempts to create a rendering device that uses the non-compliant surfaces will fail. You can use the DirectDraw method, **IDirectDraw7::GetDeviceIdentifier** to track hardware that imposes this limitation.

[Visual Basic]

0 To create a depth buffer in a Visual Basic application

1. Call the **Direct3D7.GetEnumZBufferFormats** method to retrieve a reference to a **Direct3DEnumPixelFormats** object that can enumerate the supported depth-buffer formats. Query the enumeration object for the depth-buffer pixel formats that the device supports.
2. Prepare a **DDSURFACEDESC2** type that describes a DirectDraw surface that matches the render-target surface's dimensions, includes the DDSCAPS_ZBUFFER capability flag, and uses a supported depth-buffer pixel format (retrieved in Step 2).
3. Create the surface in video memory or system memory, depending on what type of rendering device your application will use. (See note.)
4. Attach the depth buffer surface to the rendering surface using the **DirectDrawSurface7.AddAttachedSurface** method.

After creating the depth buffer surface and attaching it to the render-target surface, call **Direct3D7.CreateDevice** to create a rendering device that uses the render-target surface and its depth buffer.

Create the depth buffer in video memory—with the DDSCAPS_VIDEOMEMORY surface capability—when your application uses a hardware driver (HAL device), and in system memory—the DDSCAPS_SYSTEMMEMORY surface capability—when using software emulation drivers (the MMX or RGB devices). Failing to create a depth buffer in the appropriate type of memory will cause the **CreateDevice** method to fail.

Note

Some popular hardware devices require that the render target and depth buffer surfaces use the same bit depth. On such hardware, if your application uses a 16-bit render-target surface, the attached depth buffer must also be 16-bits. For a 32-bit render-target surface, the depth buffer must be 32-bits, of which 8-bits can be used for stencil buffering (if needed).

If the hardware upon which your application is running has this requirement, and your application fails to meet it, any attempts to create a rendering device that uses the non-compliant surfaces will fail. You can use the `DirectDraw` method, **`DirectDraw7.GetDeviceIdentifier`** and the associated object (**`DirectDrawIdentifier`**) to track hardware that imposes this limitation.

Enabling Depth Buffering

[C++]

After you create a depth buffer (as described in [Creating a Depth Buffer](#)), enabling depth buffering is as simple as calling the **`IDirect3DDevice7::SetRenderState`** method. Set the `D3DRENDERSTATE_ZENABLE` render state to enable depth-buffering. Use the `D3DZB_TRUE` value (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 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](#)), enabling depth buffering is as simple as calling the **`Direct3DDevice7.SetRenderState`** method. Set the `D3DRENDERSTATE_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 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.)

Clearing Depth Buffers

[C++]

Many C++ applications clear the depth buffer prior to rendering each new frame. You can explicitly clear the depth buffer through Direct3D by using the

IDirect3DDevice7::Clear method. The **Clear** method allows you to specify an arbitrary depth value in the **dvZ** parameter.

You can also use DirectDraw to clear a depth buffer. Call the depth buffer surface's **IDirectDrawSurface7::Blt** method to clear it. The **DDBLT_DEPTHFILL** flag indicates that the blit is being used to a clear depth buffer. When this flag is specified, the **DDBLTFX** structure passed to the **IDirectDrawSurface7::Blt** method should be initialized and have its **dwFillDepth** member set to the required depth.

If the DirectDraw device driver for a 3-D accelerator is designed to provide support for depth buffer clearing in hardware, it will report the **DDCAPS_BLTDEPTHFILL** flag and should handle **DDBLT_DEPTHFILL** blits. The destination surface of a depth-fill blit must be a depth buffer surface.

[Visual Basic]

Many Visual Basic applications clear the depth buffer prior to rendering each new frame. You can explicitly clear the depth buffer through Direct3D by using the

Direct3DDevice7.Clear method. The **Clear** method allows you to specify an arbitrary depth value in the **z** parameter.

You can also use DirectDraw to clear a depth buffer. Call the depth buffer surface's **DirectDrawSurface7.Blt** method to clear it. The **DDBLT_DEPTHFILL** flag indicates that the blit is being used to a clear depth buffer. When this flag is specified, the **DDBLTFX** type passed to the **DirectDrawSurface7.Blt** method have its **IFill** member set to the required depth.

If the DirectDraw device driver for a 3-D accelerator is designed to provide support for depth buffer clearing in hardware, it will report the **DDCAPS_BLTDEPTHFILL** flag and should handle **DDBLT_DEPTHFILL** blits. The destination surface of a depth-fill blit must be a depth buffer surface.

Changing Depth Buffer Write Access

By default, the Direct3D system is allowed to write to the depth buffer. Most applications will leave writing to the depth buffer enabled, but there are some special effects that can be achieved by not allowing the Direct3D system to write to the depth buffer.

[C++]

You can disable depth buffer writes in C++ by calling the **IDirect3DDevice7::SetRenderState** method with the *dwRenderStateType* parameter set to `D3DRENDERSTATE_ZWRITEENABLE` and the *dwRenderState* parameter set to 0.

[Visual Basic]

You can disable depth buffer writes in a Visual Basic application by calling the **Direct3DDevice7.SetRenderState** method with the *state* parameter set to `D3DRENDERSTATE_ZWRITEENABLE` and the *renderstate* parameter set to 0.

Changing Depth Buffer Comparison Functions

[C++]

By default, when depth-testing is performed on a rendering surface, the Direct3D system updates the render-target surface if the corresponding depth value (z or w) for each point is less than the value already in the depth buffer. In a C++ application, you change how the system performs comparisons on depth values by calling the **IDirect3DDevice7::SetRenderState** method with the *dwRenderStateType* parameter set to `D3DRENDERSTATE_ZFUNC`. The *dwRenderState* parameter should be set to one of the values in the **D3DCMPFUNC** enumerated type.

[Visual Basic]

By default, when depth testing is performed on a rendering surface, the Direct3D system updates the render-target surface if the corresponding depth value (z or w) for each point is less than the value already in the depth buffer. In Visual Basic, you can change how the system performs comparisons on depth values by calling the **Direct3DDevice7.SetRenderState** method with the *state* parameter set to `D3DRENDERSTATE_ZFUNC`. The *renderstate* parameter should be set to one of the values 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, then the shadow, the shadow might not be visible, or depth artifacts may be visible. You could 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

IDirect3DDevice7::SetRenderState method just before rendering them, setting the *dwRenderStateType* parameter to D3DRENDERSTATE_ZBIAS, and the *dwRenderState* parameter to a value between 0-16 inclusive. A higher z-bias value will increase the likelihood that the polygons you render will be visible when displayed with other coplanar polygons.

[\[Visual Basic\]](#)

A 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

Direct3DDevice7.SetRenderState method just before rendering them, setting the first parameter to D3DRENDERSTATE_ZBIAS, and the second parameter to a value between 0-16 inclusive. A higher z-bias value will increase 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 off sections of the rendered image so that it is 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 test the user's hardware to see if it supports stencil buffers by invoking the

IDirect3D7::EnumZBufferFormats method. The method passes depth-buffer formats, in the form of **DDPIXELFORMAT** structures, to your application. The relevant information is in the **dwZBufferBitDepth**, **dwStencilBitDepth**, **dwZBitMask**, and **dwStencilBitMask** members.

[Visual Basic]

Stencil buffer information is embedded in the z-buffer data. Your application can test the user's hardware to see if it supports stencil buffers by calling the **Direct3D7.GetEnumZBufferFormats** method, then using the returned **Direct3DEnumPixelFormats** object to retrieve information about each supported depth-buffer pixel format. The **Direct3DenumPixelFormats.GetItem** method retrieves depth-buffer formats, in the form of a **DDPIXELFORMAT** type each, to your application. The relevant information is in the **IZBufferBitDepth**, **IStencilBitDepth**, **IZBitMask**, and **IStencilBitMask** members.

How the Stencil Buffer Works

[C++]

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. Bitwise **AND** the stencil reference value with the stencil mask.
2. Bitwise **AND** 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.

Written in pseudocode, these steps would look like this:

(StencilRef & StencilMask) CompFunc (StencilBufferValue & StencilMask)

Where StencilBufferValue is the contents of the stencil buffer for the current pixel. This pseudocode uses the & 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 ignored otherwise. The default comparison behavior is to write the pixel no matter how each of the bitwise operations turn-out (D3DCMP_ALWAYS). You can change this behavior by changing the value of the D3DRENDERSTATE_STENCILFUNC render state, passing one of the members of the **D3DCMPFUNC** enumerated type to identify the desired comparison function.

[Visual Basic]

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. Bitwise **And** the stencil reference value with the stencil mask.
2. Bitwise **And** 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.

Written in pseudocode, these steps would look like this:

(StencilRef And StencilMask) CompFunc (StencilBufferValue And StencilMask)

Where 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 ignored otherwise. The default comparison behavior is to write the pixel no matter how each of the bitwise operations turn-out (D3DCMP_ALWAYS). You can change this behavior by changing the value of the D3DRENDERSTATE_STENCILFUNC render state, passing one of the members 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 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 Direct3D application. Information is divided into the following sections:

- What Are Vertex Buffers?
- Vertex Buffer Descriptions
- Device Types and Vertex Processing Requirements
- Using Vertex Buffers

What Are Vertex Buffers?

[C++]

Vertex buffers, represented by the **IDirect3DVertexBuffer7** interface, are simply 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 vertex buffer rendering methods in the **IDirect3DDevice7** 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 **Direct3DVertexBuffer7** class, are simply 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 vertex buffer rendering methods in the **Direct3DDevice7** 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 you need without re-transforming it, even with interleaved render state changes. This can be very useful when rendering models that use multiple textures: the geometry is only transformed 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*.

You can optimize geometry in vertex buffers to get maximum performance for vertex operations and rendering. See *Optimizing a Vertex Buffer* for details.

[C++]

Note

Internally, vertex buffers use *DirectDrawSurface* objects for their memory management services. As a result, the semantics for accessing vertex buffer memory are similar to those of *DirectDrawSurface* objects. In fact, the **IDirect3DVertexBuffer7::Lock** method accepts the same flags as the **IDirectDrawSurface7::Lock** method. For more information, see *Accessing the Contents of a Vertex Buffer*.

[Visual Basic]

Note

Internally, vertex buffers use *DirectDrawSurface* objects for their memory management services. As a result, the semantics for accessing vertex buffer memory are similar to those of *DirectDrawSurface* objects. In fact, the **Direct3DVertexBuffer7.Lock** method accepts the same flags as the **DirectDrawSurface7.Lock** method. For more information, see *Accessing the Contents of a Vertex Buffer*.

Vertex Buffer Descriptions

[C++]

A vertex buffer is described in terms of its capabilities: if it can only exist in system memory, if it is only to be used for write operations, the type and number of vertices it can contain, and whether or not it has been optimized since it was created. All these traits are held within a **D3DVERTEXBUFFERDESC** structure.

[Visual Basic]

A vertex buffer is described in terms of its capabilities: if it can only exist in system memory, if it is only to be used for write operations, the type and number of vertices it can contain, and whether or not it has been optimized since it was created. All these traits are held within a **D3DVERTEXBUFFERDESC** type.

Vertex buffer descriptions tell the system how to create a vertex buffer or tell your application how an existing buffer was created (and if it has been optimized since being created). You must specify a complete description to create a new vertex buffer, and you provide an empty description structure for the system to fill with the capabilities of a previously created vertex buffer. For more information about these tasks, see [Creating a Vertex Buffer and Retrieving Vertex Buffer Descriptions](#).

[C++]

The **dwSize** member, common to most DirectX structures, is used to identify structure versions and must be set to the structure size before the structure can be used with any methods. The **dwCaps** structure member contains general capability flags. The **D3DVBCAPS_SYSTEMMEMORY** flag indicates that the system should create (or already has created) the vertex buffer in system memory. Create an explicit system memory vertex buffer if your application uses a software rendering device; otherwise, it is best to let the system determine the best location by omitting the flag. For more information about explicit system memory vertex buffers, see [Device Types and Vertex Processing Requirements](#).

The presence of the **D3DVBCAPS_WRITEONLY** flag in **dwCaps** hints to the system that the vertex buffer memory is used only for write operations. This frees the driver to place the vertex data in the best possible memory location it can to enable fast processing and rendering. Reading from a vertex buffer that uses this flag can result in very slow memory access times. If the **D3DVBCAPS_WRITEONLY** flag isn't used, the driver is less likely to put the data in a location inefficient for read operations, sacrificing some processing and rendering speed. If no flags are specified, it is assumed that applications will perform read and write operations on the data within the vertex buffer.

Note

The **D3DVBCAPS_OPTIMIZED** flag is not used during vertex buffer creation. The system sets this capability when it optimizes a vertex buffer.

The final two **D3DVERTEXBUFFERDESC** structure members describe other capabilities. The **dwFVF** member contains a combination of flexible vertex format flags that identify the type of vertices that the buffer can contain. Vertex buffer capacity is measured by the total number of vertices it can contain, given in the **dwNumVertices** member.

[Visual Basic]

The **ICaps** member contains general capability flags. The **D3DVBCAPS_SYSTEMMEMORY** flag indicates that the system should create (or already has created) the vertex buffer in system memory. Create an explicit system memory vertex buffer if your application uses a software rendering device; otherwise, it is best to let the system determine the best location by omitting the flag. For more information about explicit system memory vertex buffers, see Device Types and Vertex Processing Requirements.

The presence of the **D3DVBCAPS_WRITEONLY** flag in **ICaps** hints to the system that the vertex buffer memory is used only for write operations. This frees the driver to place the vertex data in the best possible memory location it can to enable fast processing and rendering. Reading from a vertex buffer that uses this flag can result in very slow memory access times. If the **D3DVBCAPS_WRITEONLY** flag isn't used, the driver is less likely to put the data in a location inefficient for read operations, sacrificing some processing and rendering speed. If no flags are specified, it is assumed that applications will perform read and write operations on the data within the vertex buffer.

Note

The **D3DVBCAPS_OPTIMIZED** flag is not used during vertex buffer creation. The system sets this capability when it optimizes a vertex buffer.

The final two **D3DVERTEXBUFFERDESC** members describe other capabilities. The **IFVF** member contains a combination of flexible vertex format flags that identify the type of vertices that the buffer can contain. Vertex buffer capacity is measured by the total number of vertices it can contain, given in the **INumVertices** member.

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 **D3DVBCAPS_SYSTEMMEMORY** capability flag is set in the vertex buffer description, the vertex buffer is created in system memory. When the capability is not used, the device driver determines where

the memory for 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++]

Direct3D applies vertex processing operations when you call the **IDirect3DDevice7::DrawPrimitiveVB**, **IDirect3DDevice7::DrawIndexedPrimitiveVB**, or **IDirect3DVertexBuffer7::ProcessVertices** methods. These operations include transformation and lighting. The performance of these calls depends on the location of the buffer in memory, and whether that location is best for the given type of device (hardware or software).

[Visual Basic]

Direct3D applies vertex processing operations when you call the **Direct3DDevice7.DrawPrimitiveVB**, **Direct3DDevice7.DrawIndexedPrimitiveVB**, or **Direct3DVertexBuffer7.ProcessVertices** methods. These operations include transformation and lighting. The performance of these calls depends on the location of the buffer in memory, and whether that location is best for the given type of device (hardware or software).

The reasoning you should apply to determine the memory location—system or driver optimal—for vertex buffers is the same as textures. In the case of textures, hardware rasterization works best when textures are allocated in driver-optimal memory, while software rasterization works best with system-memory textures. Likewise, 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.

Note

Vertex processing done by the **ProcessVertices** method is always done in software, so the source vertex buffer for must always be in system memory. The destination vertex buffer for can be allocated in driver-optimal memory unless it is to be directly locked and read by the application, or if clipping is requested in the call to **ProcessVertices** (which requires reading from the vertex buffer when a vertex-buffer rendering method is called).

If your application performs its own vertex processing (and passes transformed, lit, and clipped vertices to the vertex-buffer rendering methods), then vertices can be directly written by the application into 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. So, if your application needs to read during processing or if writes data to the buffer erratically, then a system-memory vertex buffer is a better choice.

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. If the device is a TnLHAL Device (IID_IDirect3DTnLHALDevice), vertex processing is done in hardware and the vertex buffer should be created in driver-optimal memory for best performance. If the device is a HAL Device (IID_IDirect3DHALDevice), vertex processing is done in software and the vertex buffer should be created in system memory. When rendering with the RGB Device (IID_IDirect3DRGBDevice), all vertex buffers should be allocated in system memory.

[C++]

The following C++ style pseudo-code summarizes the logic you should apply when creating and processing vertex buffers:

```
if (software rasterization)
    use system-memory vertex buffers (D3DVBCAPS_SYSTEMMEMORY)
else if (application vertex processing)
    if (application does not read and does not use erratic writes)
        use driver-optimal vertex buffers
    else
        use system-memory vertex buffers (D3DVBCAPS_SYSTEMMEMORY)
else // Direct3D vertex processing
    if (TnLHAL device)
        use driver-optimal vertex buffers
    else if (HAL device)
        use system-memory vertex (D3DVBCAPS_SYSTEMMEMORY)
```

[Visual Basic]

The following Visual Basic style pseudo-code summarizes the logic you should apply when creating and processing vertex buffers:

```
If (software rasterization) Then
    use system-memory vertex buffers (D3DVBCAPS_SYSTEMMEMORY)
Else If (application vertex processing) Then
    If (application does not read and does not use erratic writes) Then
        use driver-optimal vertex buffers
    Else
        use system-memory vertex buffers (D3DVBCAPS_SYSTEMMEMORY)
Else // Direct3D vertex processing
    If (TnLHAL device) Then
        use driver-optimal vertex buffers
    Else If (HAL device)
        use system-memory vertex (D3DVBCAPS_SYSTEMMEMORY)
```

End If

Using Vertex Buffers

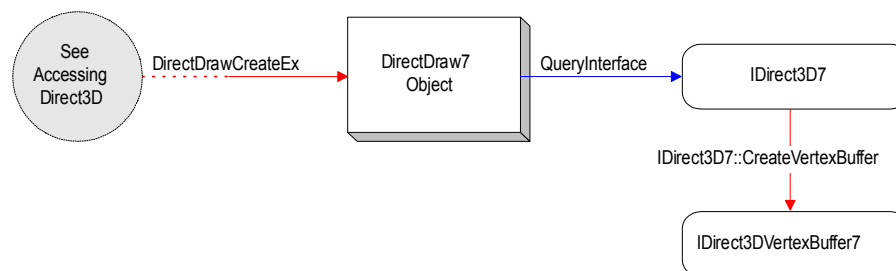
The following topics discuss common tasks that applications will perform when working with vertex buffers:

- Creating a Vertex Buffer
- Accessing the Contents of a Vertex Buffer
- Processing Vertices
- Optimizing a Vertex Buffer
- Rendering From a Vertex Buffer
- Retrieving Vertex Buffer Descriptions

Creating a Vertex Buffer

[C++]

The following figure illustrates the steps necessary to create a single vertex buffer from C++.



Note

As discussed in Accessing Direct3D, the DirectDraw component comprises two COM objects. The newest object—called DirectDraw7 and created by calling the **DirectDrawCreateEx** function—is the only object that exposes the **IDirect3D7** interface, and is the only object capable of spawning a vertex buffer that exposes the **IDirect3DVertexBuffer7** interface. The Direct3DDevice object created in this manner doesn't expose legacy Direct3D interfaces. Applications that require previous iterations of the Direct3D must use the **DirectDrawCreate** method to create a DirectDraw object and then create the desired legacy objects.

You create a vertex buffer object by calling the **IDirect3D7::CreateVertexBuffer** method, which accepts four parameters. The first parameter is the address of a **D3DVERTEXBUFFERDESC** structure that describes the desired vertex format, buffer size, and general capabilities. These capabilities are detailed in Vertex Buffer

Descriptions. Normally, the system automatically determines the best memory location (system or display memory) for the vertex buffer. However, software devices can only be used with explicit system-memory vertex buffers. For more information, see Device Types and Vertex Processing Requirements.

The second parameter that **CreateVertexBuffer** accepts is the address of a variable that will be filled with a pointer to the new **IDirect3DVertexBuffer7** interface of the vertex buffer object if the call succeeds. The third parameter determines if the vertex buffer will be capable of containing clipping information—in the form of clip flags—for vertices that exist outside the viewing area. Set this to 0 to create a "clipping-capable" vertex buffer, or include the **D3DDP_DONOTCLIP** flag to create a vertex buffer that cannot contain clip flags. The **D3DDP_DONOTCLIP** flag is only applied if you also indicate that the vertex buffer will contain transformed vertices (the **D3DFVF_XYZRHW** flag is included in the **dwFVF** member of the description structure). The **CreateVertexBuffer** method ignores the **D3DDP_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.

Note

Creating a vertex buffer that can contain clip flags does not necessarily mean that you must request that clip flags be generated during vertex processing or applied during rendering. Each vertex buffer rendering method accepts the **D3DDP_DONOTCLIP** flag to bypass clipping during rendering, and the **IDirect3DVertexBuffer7::ProcessVertices** method accepts the **D3DVOP_CLIP** flag, which can be omitted to prevent the system from generating clip flags while it processes vertices.

There is no way to produce clip flags for a vertex buffer that was created without support for them. Attempts to use the **IDirect3DVertexBuffer7::ProcessVertices** method to do this will fail in debug builds, returning **D3DERR_INVALIDVERTEXFORMAT**. Rendering methods will ignore clipping requests when rendering from a transformed vertex buffer that does not contain clip flags.

The last parameter that **CreateVertexBuffer** accepts is provided for future compatibility with COM aggregation features. Currently, aggregation isn't supported, so this parameter must be set to **NULL**.

The following C++ example shows what creating a vertex buffer might look like in code:

```
/*
 * For the purposes of this example, the g_lpD3D variable is the
 * address of an IDirect3D7 interface exposed by a Direct3D
 * object, g_vbVertexBuffer is a variable of type LPD3DVERTEXBUFFER7,
 * and the flsAHardwareDevice variable is a BOOL variable that is
 * assumed to be set during application initialization.
```

```

*/

D3DVERTEXBUFFERDESC vbdesc;
ZeroMemory(&vbdesc, sizeof(D3DVERTEXBUFFERDESC));
vbdesc.dwSize= sizeof(D3DVERTEXBUFFERDESC);
vbdesc.dwCaps      = 0L;
vbdesc.dwFVF       = D3DFVF_VERTEX;
vbdesc.dwNumVertices = NUM_FLAG_VERTICES;

// If this isn't a hardware device, make sure the
// vertex buffer uses system memory.
if( !IsAHardwareDevice )
    vbdesc.dwCaps |= D3DVCAPS_SYSTEMMEMORY;

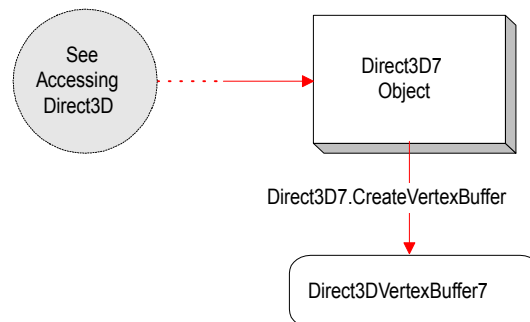
// Create a clipping-capable vertex buffer.
if(FAILED(g_lpD3D->CreateVertexBuffer(&vbdesc,
                                     &g_pvbVertexBuffer, 0L,
                                     NULL)))

    return E_FAIL;

```

[Visual Basic]

The following figure illustrates the steps necessary to create a single vertex buffer from Visual Basic.



You create a vertex buffer object from Visual Basic by calling the **Direct3D7.CreateVertexBuffer** method, which accepts four parameters. The first parameter is the address of a **D3DVERTEXBUFFERDESC** type that describes the desired vertex format, buffer size, and general capabilities. These capabilities are detailed in Vertex Buffer Descriptions. Normally, the system automatically determines the best memory location (system or display memory) for the vertex buffer. However, software devices can only be used with explicit system-memory vertex buffers. For more information, see Device Types and Vertex Processing Requirements.

The second parameter determines if the vertex buffer will be capable of containing clipping information—in the form of clip flags—for vertices that exist outside the

viewing area. Set this to 0 to create a "clipping-capable" vertex buffer, or include the D3DDP_DONOTCLIP flag to create a vertex buffer that cannot contain clip flags. The D3DDP_DONOTCLIP flag is only applied if you also indicate that the vertex buffer will contain transformed vertices (the D3DFVF_XYZRHW flag is included in the IFVF member of the description). The **CreateVertexBuffer** method ignores the D3DDP_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 **CreateVertexBuffer** method returns a reference to a new **Direct3DVertexBuffer7** object if the call succeeds.

Note

Creating a vertex buffer that can contain clip flags does not necessarily mean that you must request that clip flags be generated during vertex processing or applied during rendering. Each vertex buffer rendering method accepts the D3DDP_DONOTCLIP flag to bypass clipping during rendering, and the **Direct3DVertexBuffer7.ProcessVertices** method accepts the D3DVOP_CLIP flag, which can be omitted to prevent the system from generating clip flags while it processes vertices.

There is no way to produce clip flags for a vertex buffer that was created without support for them. Attempts to use the **Direct3DVertexBuffer7.ProcessVertices** method to do this will fail in debug builds, raising the D3DERR_INVALIDVERTEXFORMAT error. Rendering methods will ignore clipping requests when rendering from a transformed vertex buffer that does not contain clip flags.

The following Visual Basic example shows what creating a vertex buffer might look like in code:

```
'
' For the purposes of this example, the g_D3D variable is a reference
' to a Direct3D7 object and g_VertexBuffer is a reference to a
' Direct3DVertexBuffer7 object.
'

On Local Error Resume Next

Dim VBDesc As D3DVERTEXBUFFERDESC
VBDesc.IFVF = D3DFVF_VERTEX
VBDesc.INumVertices = NUM_FLAG_VERTICES

' Create a clipping-capable vertex buffer.
Set g_VertexBuffer = g_D3D.CreateVertexBuffer(VBDesc, D3DDP_DEFAULT)

If Err.Number <> DD_OK Then
    ' Code to handle error goes here.
```

End If

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 **IDirect3DVertexBuffer7::Lock** method, then access the memory as needed to fill the buffer with new vertex data, or to read any data it already contains.

The **IDirect3DVertexBuffer7::Lock** method accepts three parameters. The first, *dwFlags*, tells the system how the memory should be locked, and can be used to hint how the application will be accessing the data within the buffer. (You can hint for read-only or write-only access, which allows the driver to lock the memory to provide the best performance given the requested access type. These hints are not required, but can improve performance for memory access in some situations.) Because vertex buffers use **DirectDrawSurface** objects to contain vertex data, the flags that **IDirect3DVertexBuffer7::Lock** accepts are identical to those accepted by the **IDirectDrawSurface7::Lock** method, with identical results.

The second parameter accepted by the **Lock** method, *lppData*, is the address of an **LPVOID** variable that will contain a valid pointer to the vertex buffer memory if the call succeeds. The last parameter, *lpdwSize*, is the address of a variable that will contain the size, measured in bytes, of the buffer at *lppData* after the call returns. You can set *lpdwSize* to **NULL** if your application doesn't need information about the buffer size.

Performance Notes

Use the **DDLOCK_READONLY** flag if your application will only be reading 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. Although it is possible to write to memory locked with the **DDLOCK_READONLY** flag, doing so can produce unexpected results. In addition, attempting to read from a vertex buffer that was created with the **D3DVBCAPS_WRITEONLY** flag can be extremely slow, even if you lock the buffer for read-only access.

In C++, you directly access the memory allocated for the vertex buffer. As such, you must take care that your application properly accesses the allocated memory, or risk rendering that memory invalid. Use the stride of the vertex format your application uses to move from one vertex in the allocated buffer to another. The vertex buffer memory itself is a simple array of vertices, specified in flexible vertex format. If your application uses the legacy vertex structures, **D3DVERTEX**, **D3DLVERTEX**, and **D3DTLVERTEX**, the stride is simply the size of the structure, in bytes. If you are using a vertex format different from the legacy formats, 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 within 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) × 2 × n
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). Because each set of texture coordinates in the vertex format occupies the space of two **float** variables, multiply the number of texture coordinate sets by the size of one set of texture coordinates 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 Visual Basic applications to easily update the vertex data they contain. You prepare to access the contents of the vertex buffer by calling the **Direct3DVertexBuffer7.Lock** method. The **Lock** method offers the *flags* parameter to tell the system how the memory should be locked. The *flags* parameter hints to the system how the application will be accessing the data within the buffer. (You can request read-only or write-only access, which allows the driver to lock the memory to provide the best performance given the requested access type. These hints are not required, but can improve performance for memory access in some situations.) Because vertex buffers use DirectDrawSurface objects to contain vertex data, the flags that **Direct3DVertexBuffer7.Lock** accepts are identical to those accepted by the **DirectDrawSurface7.Lock** method, with identical results.

Performance Notes

Use the DDLOCK_READONLY flag if your application will only be reading 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. Although it is possible to write to memory locked with the DDLOCK_READONLY flag, doing so can produce unexpected results. In addition, attempting to read from a vertex buffer that was created with the D3DVBCAPS_WRITEONLY flag can be extremely slow, even if you lock the buffer for read-only access.

Once the vertex buffer is locked, you can call the **Direct3DVertexBuffer7.GetVertices** method to fill an array with the current

contents of the vertex buffer. Set the first parameter to the zero-based index of the first vertex in the buffer to need and set the second parameter to the total number of vertices to retrieve. The last parameter should be the first element in an array that uses the same vertex format as the vertex buffer, which will contain the requested data after the call.

The vertex buffer memory itself 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). Because each set of texture coordinates in the vertex format occupies the space of two **Single** variables, multiply the number of texture coordinate sets by the size of one set of texture coordinates to calculate the memory required for that number of texture coordinates.

To change the contents of the vertex buffer after it is locked, modify the contents of the array, then call the **Direct3DVertexBuffer7.SetVertices** method to copy the vertices into the buffer. You must also unlock the vertex buffer by way of the **Direct3DVertexBuffer7.Unlock** method before performing any vertex buffer processing or rendering operations.

Processing Vertices

[C++]

Processing the vertices in a vertex buffer applies the current transformation matrices for the device, and can optionally apply vertex operations such as lighting, generating clip flags, and updating extents. The **IDirect3DVertexBuffer7** interface exposes the **IDirect3DVertexBuffer7::ProcessVertices** method to process vertices. You process vertices from a source vertex buffer into a destination vertex buffer by calling the **ProcessVertices** method of the destination vertex buffer, not the source buffer. The method accepts seven parameters that describe the operations to be performed, the location of the source vertex buffer's **IDirect3DVertexBuffer7** interface, the rendering device that will perform the vertex operations, and the location and quantity of vertices that the method targets. After the call, the destination buffer contains the processed vertex data, and the source buffer is unchanged.

When preparing to process vertices, set the first parameter, *dwVertexOp*, to indicate the vertex operations you want to perform. You must include the D3DVOP_TRANSFORM flag, or the method will fail, but the remaining operations are optional. You can include any combination of optional flags to light the vertices, generate clip flags, and update extents during vertex processing.

The second and third parameters, *dwDestIndex* and *dwCount*, reflect the index within the destination buffer at which the vertices will be placed and the total number of vertices that should be processed and placed in the destination buffer. The fourth parameter, *lpSrcBuffer*, should be set to the address of the **IDirect3DVertexBuffer7** of the vertex buffer object that contains the source vertices. The *dwSrcIndex* specifies the index at which the method should start processing vertices. (The total number of

source vertices to be processed is implied by the *dwCount* parameter.) Set the *lpD3DDevice* parameter to the address of the **IDirect3DDevice7** interface for the rendering device that processes the vertices.

The final parameter, *dwFlags*, 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 *dwFlags* 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 source vertex buffer (when present in the source buffer) or set to 0. However, by using **D3DPV_DONOTCOPYDATA**, any vertex components of the destination buffer's vertex format not computed as part of the call to **ProcessVertices** are left untouched. Picture yourself calling the **ProcessVertices** method intending to transform vertices without lighting them (with the **D3DVOP_TRANSFORM** vertex operation by itself). If the destination vertex buffer uses the **D3DFVF_TLVERTEX** vertex format, specifying **D3DPV_DONOTCOPYDATA** flag will cause the system to ignore the diffuse, specular, and the texture coordinate components of the destination buffer. The result is lighter processing and data transfer overhead—you only wanted the system to transform the vertices. Without using **D3DPV_DONOTCOPYDATA**, the system would copy the extraneous lighting and texture coordinate components to the vertices in the destination buffer, even though your application doesn't need them. The **D3DPV_DONOTCOPYDATA** flag is useful to reduce the amount of work required by the system for vertex processing. An application that uses **ProcessVertices** can copy static vertex components to the destination vertex buffer directly, eliminating the need for the system to copy from the source buffer to the destination buffer.

[Visual Basic]

Processing the vertices in a vertex buffer applies the current transformation matrices for the device, and can optionally apply vertex operations such as lighting, generating clip flags, and updating extents. The **Direct3DVertexBuffer7** class offers the **Direct3DVertexBuffer7.ProcessVertices** method to process vertices. You process vertices from a source vertex buffer into a destination vertex buffer by calling the **ProcessVertices** method of the destination vertex buffer, not the source buffer. The method accepts seven parameters that describe the operations to be performed, a reference to the source **Direct3DVertexBuffer7** object, the rendering device that will perform the vertex operations, and the location and quantity of vertices that the method targets. After the call, the destination buffer contains the processed vertex data, and the source buffer is unchanged.

When preparing to process vertices, set the first parameter, *vertexOp*, to indicate the vertex operations you want to perform. You must include the **D3DVOP_TRANSFORM** flag, or the method will fail, but the remaining operations are optional. You can include any combination of optional flags to light the vertices, generate clip flags, and update extents during vertex processing.

The second and third parameters, *destIndex* and *count*, reflect the index within the destination buffer at which the vertices will be placed and the total number of vertices

that should be processed and placed in the destination buffer. The fourth parameter, *srcBuffer*, should be the **Direct3DVertexBuffer7** object that contains the source vertices. The *srcIndex* specifies the index at which the method should start processing vertices. (The total number of source vertices to be processed is implied by the *count* parameter.) Set the *dev* parameter to the **Direct3DDevice7** object that will process the 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 source vertex buffer (when present in the source buffer) or set to 0. However, by using **D3DPV_DONOTCOPYDATA**, any vertex components of the destination buffer's vertex format not computed as part of the call to **ProcessVertices** are left untouched. Picture yourself calling the **ProcessVertices** method intending to transform vertices without lighting them (with the **D3DVOP_TRANSFORM** vertex operation by itself). If the destination vertex buffer uses the **D3DFVF_TLVERTEX** vertex format, specifying **D3DPV_DONOTCOPYDATA** flag will cause the system to ignore the diffuse, specular, and the texture coordinate components of the destination buffer. The result is lighter processing and data transfer overhead—you only wanted the system to transform the vertices. Without using **D3DPV_DONOTCOPYDATA**, the system would copy the extraneous lighting and texture coordinate components to the vertices in the destination buffer, even though your application doesn't need them. The **D3DPV_DONOTCOPYDATA** flag is useful to reduce the amount of work required by the system for vertex processing. An application that uses **ProcessVertices** can copy static vertex components to the destination vertex buffer directly, eliminating the need for the system to copy from the source buffer to the destination buffer.

Take care to create vertex buffers that use compatible vertex formats. At the least, the source buffer should contain untransformed vertices (using the **D3DFVF_XYZ** flag in the vertex format of the buffer description), and the destination buffer should contain transformed vertices (using the **D3DFVF_XYZRHW** flag). Any lighting or clipping services require that the source and destination vertex formats contain the appropriate fields. For instance, don't request lighting on vertices when the vertex format doesn't include a vertex normal. Likewise, you can't request that the system produce clip flags for a destination vertex buffer that was created without clipping capabilities. Attempts to perform operations on incompatible buffers will fail in debug builds.

You cannot process vertices when the source or destination vertex buffers are locked.

Optimizing a Vertex Buffer

Optimizing a vertex buffer causes the system to modify the contents of the buffer for better performance when processing or rendering vertices. Exactly how vertices are

optimized is platform-specific and subject to change. Consequently, you cannot lock or otherwise access the contents of an optimized vertex buffer.

[C++]

Performance of both hardware and software devices can benefit from using optimized vertex buffers. The optimization of vertex buffers differs depending on whether the vertex buffer is to be used for hardware or software vertex processing. Vertex buffers created with the `D3DVBCAPS_SYSTEMMEMORY` flag will, when optimized, be optimized for use with software vertex processing only, and can only be used with the **IDirect3DVertexBuffer7::ProcessVertices** method or the vertex-buffer rendering methods on a HAL Device (as opposed to a TnLHAL Device). Vertex buffers created without the `D3DVBCAPS_SYSTEMMEMORY` flag (for driver-optimal memory) will be optimized for hardware vertex processing, and thus can only be used for rendering methods on a TnLHAL Device. Optimized vertex buffers cannot be used as the destination buffer for the **IDirect3DVertexBuffer7::ProcessVertices** or **IDirect3DVertexBuffer7::ProcessVerticesStrided** methods.

Call the **IDirect3DVertexBuffer7::Optimize** method to optimize a vertex buffer. Optimizing a vertex buffer improves performance of vertex operations and rendering. The **IDirect3DVertexBuffer7::Optimize** method accepts two parameters, only one of which is used. Set the *lpD3DDevice* parameter to the address of the device for which the vertices will be optimized, and set the last parameter to 0. Locked vertex buffers cannot be optimized until they are unlocked.

You can improve performance by keeping vertices in optimized format. However, optimized vertex buffers should be used only for static geometry, because once a vertex buffer is optimized, you can't lock it to change the optimized contents. After you optimize a vertex buffer, it can only be used with the **IDirect3DVertexBuffer7::ProcessVertices** method and the vertex buffer rendering methods.

[Visual Basic]

Performance of both hardware and software devices can benefit from using optimized vertex buffers. The optimization of vertex buffers differs depending on whether the vertex buffer is to be used for hardware or software vertex processing. Vertex buffers created with the `D3DVBCAPS_SYSTEMMEMORY` flag will, when optimized, be optimized for use with software vertex processing only, and can only be used with the **Direct3DVertexBuffer7.ProcessVertices** method or the vertex-buffer rendering methods on a HAL Device (as opposed to a TnLHAL Device). Vertex buffers created without the `D3DVBCAPS_SYSTEMMEMORY` flag (for driver-optimal memory) will be optimized for hardware vertex processing, and thus can only be used for rendering methods on a TnLHAL Device. Optimized vertex buffers cannot be used as the destination buffer for the **Direct3DVertexBuffer7.ProcessVertices** method.

Call the **Direct3DVertexBuffer7.Optimize** method to optimize a vertex buffer. Optimizing a vertex buffer improves performance of vertex operations and rendering. The **Direct3DVertexBuffer7.Optimize** method accepts one parameter. Set it to the

Direct3DDevice7 object for which the vertices will be optimized. Locked vertex buffers cannot be optimized until they are unlocked.

You can improve performance by keeping vertices in optimized format. However, optimized vertex buffers should be used only for static geometry, because once a vertex buffer is optimized, you can't lock it to change the optimized contents. After you optimize a vertex buffer, it can only be used with the **Direct3DVertexBuffer7.ProcessVertices** method and the vertex buffer rendering methods.

Rendering From a Vertex Buffer

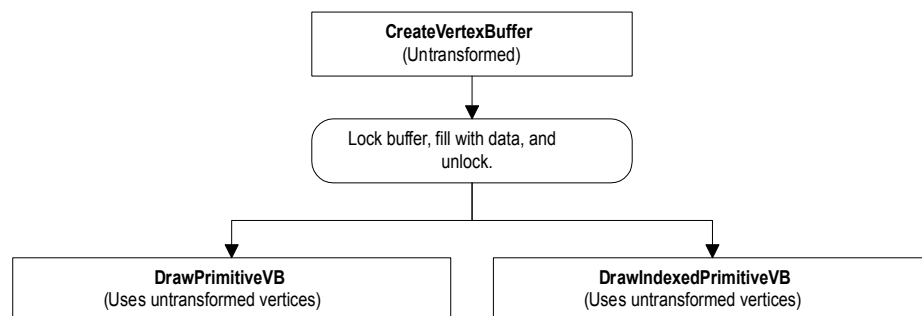
As introduced in Rendering, rendering devices offer methods to render primitives from a vertex buffer. These methods work very much like most other rendering methods, but employ slightly different parameters to accommodate vertex buffer objects.

The following topics introduce common rendering situations, and discuss specific issues related to calling the vertex buffer rendering methods:

- About Vertex Buffer Rendering
- Calling Vertex Buffer Rendering Methods

About Vertex Buffer Rendering

There are two common situations in which your application will render vertices from a vertex buffer. At the most basic level, the two situations break down according to the type of vertex that is in the vertex buffer at "render time," but indirectly, the procedure your application uses also determines when vertex and rendering operations occur. The following diagram illustrates the process of rendering from an untransformed vertex buffer.



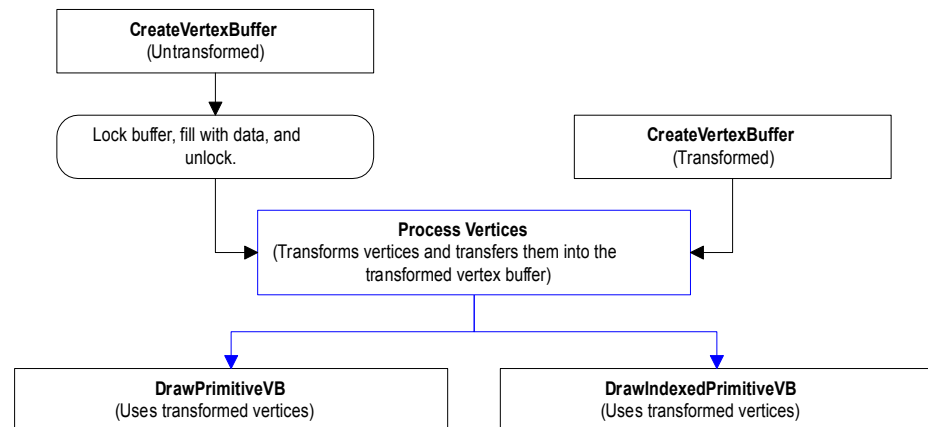
[C++]

As shown in the preceding illustration, the **IDirect3DDevice7::DrawPrimitiveVB** and **IDirect3DDevice7::DrawIndexedPrimitiveVB** methods are capable of rendering from a non-transformed vertex buffer. In this case, the system performs

vertex and rendering operations each time you call a rendering method. Using this approach isn't likely to provide improved performance over traditional `DrawPrimitive` rendering methods, but it might be more convenient in some situations. You can optimize performance by reusing transformed vertex data when you can, as shown in the following illustration.

[Visual Basic]

As shown in the preceding illustration, the **`Direct3DDevice7.DrawPrimitiveVB`** and **`Direct3DDevice7.DrawIndexedPrimitiveVB`** Visual Basic methods are capable of rendering from a non-transformed vertex buffer. In this case, the system performs vertex and rendering operations each time you call a rendering method. Using this approach isn't likely to provide improved performance over traditional `DrawPrimitive` rendering methods, but it might be more convenient in some situations. You can optimize performance by reusing transformed vertex data when you can, as shown in the following illustration.



[C++]

In this case, your application creates two vertex buffers: one for untransformed geometry, and another for transformed geometry. The second buffer receives transformed vertex data when the **`IDirect3DVertexBuffer7::ProcessVertices`** is called. The **`ProcessVertices`** method reads the vertices in the source buffer, performs the requested vertex operations on them, and places the results in the destination buffer. You can call the same rendering methods for transformed vertices that you would to render untransformed vertices. However, unlike untransformed vertices, Direct3D automatically detects that the data in the vertex buffer is transformed, sending it to be rasterized right away. The performance overhead is kept to a minimum by eliminating unnecessary transformations.

Note

The **IDirect3DVertexBuffer7** interface provides a new method, **IDirect3DVertexBuffer7::ProcessVerticesStrided**, to process strided vertices into a destination vertex buffer.

[Visual Basic]

In this case, your application creates two vertex buffers: one for untransformed geometry, and another for transformed geometry. The second buffer receives transformed vertex data when the **Direct3DVertexBuffer7.ProcessVertices** is called. The **ProcessVertices** method reads the vertices in the source buffer, performs the requested vertex operations on them, and places the results in the destination buffer. You can call the same rendering methods for transformed vertices that you would to render untransformed vertices. However, unlike untransformed vertices, Direct3D automatically detects that the data in the vertex buffer is transformed, sending it be rasterized right away. The performance overhead is kept to a minimum by eliminating unnecessary transformations.

You can optimize vertex buffer contents to increase performance even more. For more information, see *Optimizing a Vertex Buffer*.

Calling Vertex Buffer Rendering Methods

[C++]

The **IDirect3DDevice7::DrawPrimitiveVB** method corresponds to the **IDirect3DDevice7::DrawPrimitive** method. Like its relative, the **DrawPrimitiveVB** method assumes that vertices appear in sequential order within the vertex buffer.

The **DrawPrimitiveVB** method accepts five parameters. Set the *d3dptPrimitiveType* parameter to indicate the type of primitive being rendered by using one of the members of the **D3DPRIMITIVETYPE** enumerated type. Specify the address of the vertex buffer that contains the vertices in the second parameter, *lpd3dVertexBuffer*, then set values in the *dwStartVertex* and *dwNumVertices* parameters to reflect the first vertex and the total number of vertices that will be rendered. You don't need to render all the vertices that the vertex buffer contains, but you must use the appropriate number of vertices for that primitive type (as documented in the reference information for **D3DPRIMITIVETYPE**). The last parameter, *dwFlags*, determines the rendering behavior. These flags are identical to those used with other rendering methods. You can set flags to enable lighting and clipping, or to update the clip extents during rendering.

The **IDirect3DDevice7::DrawIndexedPrimitiveVB** method renders primitives by indexing the vertices within a vertex buffer. The first two parameters are identical to those of the **DrawPrimitiveVB** method. The third and fourth parameters should be the index of the first vertex in the buffer and the total number of vertices to be rendered. The fifth parameter, *lpwIndices*, should be set to the address of an ordered array of **WORD** indices that the method will use to access the vertices to be rendered, and the sixth parameter, *dwIndexCount*, should be set to the number of index values

that the array contains. The *dwFlags* parameter is identical to its counterpart in the **DrawPrimitiveVB** method.

[Visual Basic]

The **Direct3DDevice7.DrawPrimitiveVB** method corresponds to the **Direct3DDevice7.DrawPrimitive** method. Like its relative, the **DrawPrimitiveVB** method assumes that vertices appear in sequential order within the vertex buffer.

The **DrawPrimitiveVB** method accepts five parameters. Set the *d3dpt* parameter to indicate the type of primitive being rendered by using one of the members of the **CONST_D3DPRIMITIVETYPE** enumeration. Specify the **Direct3DVertexBuffer7** object that contains the vertices in the second parameter, *vertexBuffer*, then set values in the *startVertex* and *numVertices* parameters to reflect the first vertex and the total number of vertices that will be rendered. You don't need to render all the vertices that the vertex buffer contains, but you must use the appropriate number of vertices for that primitive type (as documented in the reference information for **D3DPRIMITIVETYPE**). The last parameter, *flags*, determines the rendering behavior. These flags are identical to those used with other rendering methods. You can set flags to enable lighting and clipping, or to update the clip extents during rendering.

The **Direct3DDevice7.DrawIndexedPrimitiveVB** method renders primitives by indexing the vertices within a vertex buffer. The first two parameters are identical to those of the **DrawPrimitiveVB** method. The third and fourth parameters should be the index of the first vertex in the buffer and the total number of vertices to be rendered. The fifth parameter, *indexArray*, should be set to the first element in an ordered array of **Integer** indices that the method will use to access the vertices to be rendered, and the sixth parameter, *indexCount*, should be to the number of index values that the array contains. The *flags* parameter is identical to its counterpart in the **DrawPrimitiveVB** method.

You cannot request that vertex buffer rendering methods perform lighting if the format of the vertices within the buffer does not contain a vertex normal. Nor can you request that the rendering methods clip vertices when rendering from a transformed vertex buffer created with the **D3DDP_DONOTCLIP** flag. (This is allowed for untransformed vertex buffers, as the system will create a clipping-capable version at render time.) Attempts to request unavailable services will not cause the method to fail in debug builds, but the services will not be applied.

A software device, as opposed to a hardware-accelerated device, can only render from vertex buffers that were created with the **D3DVBCAPS_SYSTEMMEMORY** flag. For more information, see Device Types and Vertex Processing Requirements.

Retrieving Vertex Buffer Descriptions

[C++]

Applications often need to retrieve information about existing vertex buffers at run time. Direct3D makes this possible through the

IDirect3DVertexBuffer7::GetVertexBufferDesc method. Call this method using the **IDirect3DVertexBuffer7** interface of any vertex buffer to retrieve its current description.

The **GetVertexBufferDesc** method accepts only one parameter: the address of a properly initialized **D3DVERTEXBUFFERDESC** structure. To initialize the structure, set the **dwSize** member to the structure size, in bytes, and set the remaining members to 0. After the method returns, the structure will contain information about the vertex buffer capabilities, the vertex format in the buffer, and the total number of vertices that it can contain.

Note

Optimizing a vertex buffer sets the **D3DVBCAPS_OPTIMIZED** capability flag in the **dwCaps** member of the **D3DVERTEXBUFFERDESC** structure. If this flag is set, the vertex buffer cannot be locked. The contents of an optimized vertex buffer are not made available to anything but rendering methods and vertex buffer processing methods (**IDirect3DVertexBuffer7::ProcessVertices** and **IDirect3DVertexBuffer7::ProcessVerticesStrided**).

For general information, see Vertex Buffer Descriptions.

[Visual Basic]

Applications often need to retrieve information about existing vertex buffers at run time. Direct3D makes this possible for Visual Basic applications through the

Direct3DVertexBuffer7.GetVertexBufferDesc method.

The **GetVertexBufferDesc** method accepts only one parameter: a properly variable of type **D3DVERTEXBUFFERDESC**. After the method returns, the structure will contain information about the vertex buffer capabilities, the vertex format in the buffer, and the total number of vertices that it can contain.

Note

Optimizing a vertex buffer sets the **D3DVBCAPS_OPTIMIZED** capability flag in the **lCaps** member of the **D3DVERTEXBUFFERDESC** type retrieved by **GetVertexBufferDesc**. If this flag is set, the vertex buffer cannot be locked. The contents of an optimized vertex buffer are not made available to anything but rendering methods and the **Direct3DVertexBuffer7.ProcessVertices** method.

For general information, see Vertex Buffer Descriptions.

Common Techniques and Special Effects

Direct3D provides a powerful set of tools that can be used to increase the realistic appearance of a 3-D scene. This section presents information on some common special effects that can be produced with Direct3D. The range of possible effects is by no means limited to those presented here. The discussion in this section is organized into the following topics:

- Fog
- Geometry Blending
- User-defined Clip Planes
- Environment Mapping
- Bump Mapping
- Billboarding
- Clouds, Smoke, and Vapor Trails
- Fire, Flares, Explosions, and More
- Motion Blur
- Stencil Buffer Techniques
- Colored Lights
- Antialiasing

Fog

The following topics introduce fog and present information about using various fog features in Direct3D applications:

- Introduction to Fog
- Fog Formulas
- Fog Parameters
- Fog Blending
- Fog Color
- Pixel Fog
- Vertex Fog

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. Direct3D Immediate Mode 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 into the chosen fog color, creating the illusion that the object is being increasingly obscured by tiny particles floating in the scene—fog. The following screen capture shows a scene rendered without fog, and a similar scene rendered with fog enabled.

In the preceding figure, 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 applications with two ways to add fog to a scene, pixel fog and vertex fog, named according to 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 will be 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 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 itself is computed varies on the projection matrix or if range-based fog is enabled. For more information, see Eye-Relative vs. Z-based Depth and Range-Based Fog.

[Visual Basic]

Applications written in Visual Basic can control how fog affects the color of objects in a scene by changing how 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 varies on the projection matrix or if 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 the 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) within a scene. Values for *d* increase as objects become more distant. The linear formula is supported for both pixel fog and vertex fog, but the exponential formulas are currently only supported when using pixel fog:

D3DFOG_EXP

$$f = \frac{1}{e^{d \times density}}$$

D3DFOG_EXP2

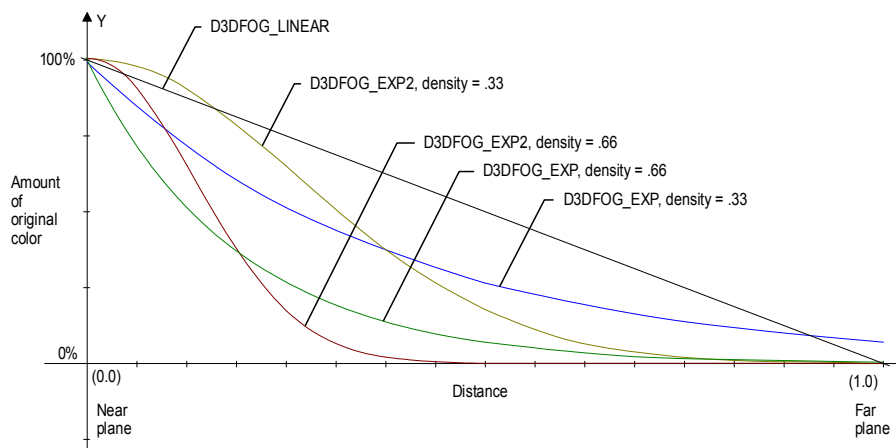
$$f = \frac{1}{e^{(d \times density)^2}}$$

In the preceding 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) within 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 DirectX 7.0. 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 of the fog formulas introduced in Fog Formulas. The

D3DFOGMODE enumerated type defines constants that you can use to identify the fog formula you want Direct3D to use. The

D3DRENDERSTATE_FOGTABLEMODE render state controls the fog mode that Direct3D will use for pixel fog, and the **D3DRENDERSTATE_FOGVERTEXMODE** render state controls the mode for vertex fog.

When using the linear fog formula, you set the starting and ending distances through the **D3DRENDERSTATE_FOGSTART** and **D3DRENDERSTATE_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 Type	Fog Start/End Units
Pixel (Z)	Device space [0.0,1.0]
Pixel (W)	Camera space
Vertex	Camera space

The **D3DRENDERSTATE_FOGDENSITY** render state controls the fog density applied when an exponential fog formula is enabled. Fog density is a essentially 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 **D3DRENDERSTATE_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 of the fog formulas introduced in Fog Formulas. The

CONST_D3DFOGMODE enumeration defines constants that you can use to identify the fog formula you want Direct3D to use. The **D3DRENDERSTATE_FOGTABLEMODE** render state controls the fog mode that Direct3D will use for pixel fog, and the **D3DRENDERSTATE_FOGVERTEXMODE** render state controls the mode for vertex fog.

When using the linear fog formula, you set the starting and ending distances through the **D3DRENDERSTATE_FOGSTART** and **D3DRENDERSTATE_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 Type	Fog Start/End Units
Pixel (Z)	Device space [0.0,1.0]
Pixel (W)	Camera space
Vertex	Camera space

The **D3DRENDERSTATE_FOGDENSITY** render state controls the fog density applied when an exponential fog formula is enabled. Fog density is a essentially 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 **D3DRENDERSTATE_FOGCOLOR** device render state. Use the **DirectX7.CreateColorRGB** method to generate valid color 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 **D3DRENDERSTATE_FOGENABLE** render state controls fog blending. Set this render state to **TRUE** to enable fog blending (the default is **FALSE**), as in the following example code:

```
//
// For this example, g_lpDevice is a valid pointer
// to an IDirect3DDevice7 interface.
HRESULT hr;
hr = g_lpDevice->SetRenderState(
    D3DRENDERSTATE_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 D3DRENDERSTATE_FOGENABLE render state controls fog blending. Set this render state to True to enable fog blending (the default is False), as in the following example code:

```
'  
' For this example, g_d3dDevice contains a valid  
' reference to an IDirect3DDevice7 object.  
'  
  
On Local Error Resume Next  
  
Call g_d3dDevice.SetRenderState( _  
    D3DRENDERSTATE_FOGENABLE, _  
    True)  
  
If Err.Number <> DD_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 D3DRENDERSTATE_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 g_lpD3DDevice variable is  
 * a valid pointer to an IDirect3DDevice7 interface.  
 */  
HRESULT hr;  
  
hr = g_lpD3DDevice->SetRenderState(  
    D3DRENDERSTATE_FOGCOLOR,  
    0x00FFFFFF); // Highest 8 bits aren't used.
```



```
if(FAILED(hr))  
    return hr;
```

[Visual Basic]

Fog color for both pixel and vertex fog is set through the `D3DRENDERSTATE_FOGCOLOR` render state. The render state values can be any RGB color, as returned by the **DirectX7.CreateColorRGB** method.

The following Visual Basic example sets the fog color to white:

```
'  
' For this example, the g_d3dDevice variable is  
' a valid reference to a Direct3DDevice7 object,  
' and g_dx is a valid reference to a DirectX7 object.  
'  
On Local Error Resume Next  
  
Call g_d3dDevice.SetRenderState(_  
    D3DRENDERSTATE_FOGCOLOR, _  
    g_dx.CreateColorRGB(1#, 1#, 1#))  
  
If Err.Number <> DD_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 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. (This is unlike vertex fog, in which 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 be applied in blending computations.

[C++]

Pixel fog can be applied using any of the fog formulas 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 of the fog formulas 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 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 homogenous coordinate set. (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 **dwRasterCaps** member of the **D3DPRIMCAPS** when you call **IDirect3DDevice7::GetCaps** method. (The **D3DDEVICEDESC7** structure you pass to **GetCaps** contains multiple **D3DPRIMCAPS** structures that describe capabilities for various types of primitives.)

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 will automatically use eye-relative depth in favor of 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 **IDirect3DDevice7::SetTransform** method, using the **D3DTRANSFORMSTATE_PROJECTION** value and passing a **D3DMATRIX** structure that represents the desired matrix.) If the projection matrix isn't compliant with this requirement, fog effects will not be 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.)

Usage Notes

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.

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.

[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 homogenous coordinate set. (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 **IRasterCaps** member of the **D3DPRIMCAPS** type when you call **Direct3DDevice7.GetCaps** method. (The **D3DDEVICEDESC7** type you pass to **GetCaps** contains multiple **D3DPRIMCAPS** type that describe capabilities for various types of primitives.)

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 will automatically use eye-relative depth in favor of 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 **Direct3DDevice7.SetTransform** method, using the D3DTRANSFORMSTATE_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 will not be 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.)

Usage Notes

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.

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++]

0 To enable pixel fog in a C++ application

1. Enable fog blending by setting the D3DRENDERSTATE_FOGENABLE render state to TRUE.
2. Set the desired fog color in the D3DRENDERSTATE_FOGCOLOR render state.
3. Choose the fog formula you want to use by setting the D3DRENDERSTATE_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_lpDevice is a valid
// pointer to an IDirect3DDevice7 interface.
void SetupPixelFog(DWORD dwColor, DWORD dwMode)
{
    float fStart = 0.5f, // for linear mode
          fEnd   = 0.8f,
          fDensity = 0.66; // for exponential modes

    // Enable fog blending.
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGENABLE, TRUE);

    // Set the fog color.
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGCOLOR, dwColor);

    // Set fog parameters.
    if(D3DFOG_LINEAR == dwMode)
    {
        g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGTABLEMODE, dwMode);
        g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGSTART, *(DWORD *)&fStart);
        g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGEND,   *(DWORD *)&fEnd);
    }
    else
    {
        g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGTABLEMODE, dwMode);
```

```

        g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGDENSITY, *(DWORD *)
(&fDensity));
    }

}

```

Note

Some fog parameters are required as floating-point values, even though the **IDirect3DDevice7::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, then dereferencing them.

[Visual Basic]

U To enable pixel fog in a Visual Basic application

1. Enable fog blending by setting the D3DRENDERSTATE_FOGENABLE render state to True.
2. Set the desired fog color in the D3DRENDERSTATE_FOGCOLOR render state.
3. Choose the fog formula you want to use by setting the D3DRENDERSTATE_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, g_d3dDevice is a valid
' pointer to a Direct3DDevice7 object.
Sub SetupPixelFog(lColor 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 g_d3dDevice.SetRenderState(D3DRENDERSTATE_FOGENABLE, True)

' Set the fog color.
Call g_d3dDevice.SetRenderState(D3DRENDERSTATE_FOGCOLOR, IColor)

' Set fog parameters.
If Mode = D3DFOG_LINEAR Then
    Call g_d3dDevice.SetRenderState(D3DRENDERSTATE_FOGTABLEMODE, Mode)
    Call g_d3dDevice.SetRenderStateSingle(D3DRENDERSTATE_FOGSTART, StartFog)
    Call g_d3dDevice.SetRenderStateSingle(D3DRENDERSTATE_FOGEND, EndFog)
Else
    Call g_d3dDevice.SetRenderState(D3DRENDERSTATE_FOGTABLEMODE, Mode)
    Call g_d3dDevice.SetRenderStateSingle(D3DRENDERSTATE_FOGDENSITY, Density)
End If
End Sub
```

Vertex Fog

This section introduces vertex fog and provides details about using it in 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, then interpolates the results across the face of the polygon during rasterization. Vertex fog effects are computed by the Direct3D lighting and transformation engine. For more information, see Fog Parameters.

If your application doesn't use Direct3D for transformation and lighting, it must perform fog calculations on its own. In this case, your application can place the fog factor it computes in the alpha component of the specular color for each vertex. You are free to use whatever formulas you desire—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

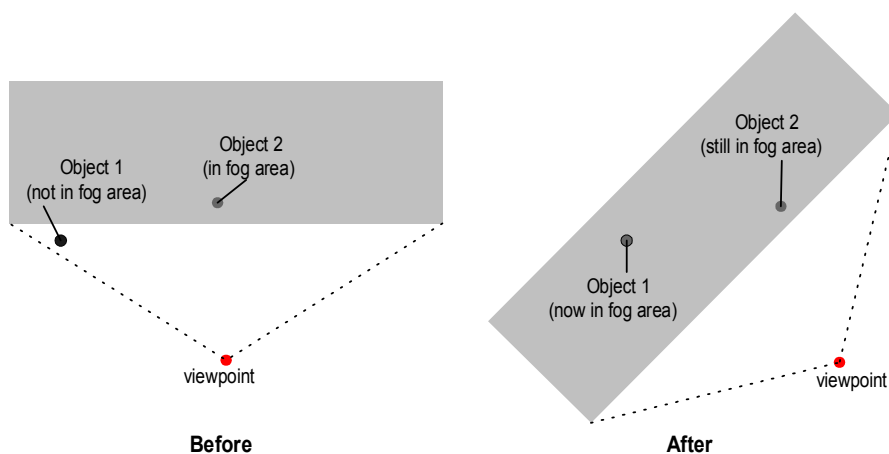
Applications that perform their own transformation and lighting must also perform their own vertex fog calculations. As a result, such an application need only enable fog blending and set the fog color through the associated render states, as described in Fog Blending and Fog Color.

Range-Based Fog

Note

Direct3D used 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 non-intuitive 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, increasing the effects 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 `D3DPRASERCAPS_FOGRANGE` capability flag in the `dwRasterCaps` member of the `D3DPRIMCAPS` structure when you call the `IDirect3DDevice7::GetCaps` method. To enable range-based fog, set the `D3DRENDERSTATE_RANGEFOGENABLE` render state to `TRUE`.

[Visual Basic]

If the current device supports range-based fog, it will set the D3DPRASERCAPS_FOGRANGE capability flag in the **IRasterCaps** member of the **D3DPRIMCAPS** type when you call the **Direct3DDevice7.GetCaps** method. To enable range-based fog, set the D3DRENDERSTATE_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++]

0 To enable vertex fog in a C++ application

1. Enable fog blending by setting D3DRENDERSTATE_FOGENABLE to TRUE.
2. Set the fog color in the D3DRENDERSTATE_FOGCOLOR render state.
3. Choose the desired fog formula by setting the D3DRENDERSTATE_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_lpDevice is a valid
// pointer to an IDirect3DDevice7 interface.
void SetupVertexFog(DWORD dwColor, DWORD dwMode, BOOL fUseRange, FLOAT
fDensity)
{
    float fStart = 0.5f, // linear fog distances
        fEnd = 0.8f;

    // Enable fog blending.
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGENABLE, TRUE);

    // Set the fog color.
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGCOLOR, dwColor);
```



```
// Set fog parameters.
if(D3DFOG_LINEAR == dwMode)
{
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGVERTEXMODE, dwMode);
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGSTART, *(DWORD *)&fStart);
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGEND, *(DWORD *)&fEnd);
}
else
{
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGVERTEXMODE, dwMode);
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGDENSITY, *(DWORD *)
(&fDensity));
}

// Enable range-based fog if desired (only supported for vertex fog).
// For this example, it is assumed that fUseRange 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(fUseRange)
    g_lpDevice->SetRenderState(
        D3DRENDERSTATE_RANGEFOGENABLE,
        TRUE);
}
```

Note

Some fog parameters are required as floating-point values, even though the **IDirect3DDevice7::SetRenderState** method only accepts **DWORD** values in the second parameter. The preceding 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, then dereferencing them.

[Visual Basic]

U To enable vertex fog from a Visual Basic application

1. Enable fog blending by setting **D3DRENDERSTATE_FOGENABLE** to **True**.
2. Set the fog color in the **D3DRENDERSTATE_FOGCOLOR** render state.
3. Choose the desired fog formula by setting the **D3DRENDERSTATE_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 Visual Basic example code 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_d3dDevice is a valid
' reference to a Direct3DDevice7 object.
Sub SetupVertexFog(lColor As Long, Mode As CONST_D3DFOGMODE, _
    bUseRange 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 g_d3dDevice.SetRenderState(D3DRENDERSTATE_FOGENABLE, True)

    ' Set the fog color.
    Call g_d3dDevice.SetRenderState(D3DRENDERSTATE_FOGCOLOR, lColor)

    ' Set fog parameters.
    If Mode = D3DFOG_LINEAR Then
        Call g_d3dDevice.SetRenderState(D3DRENDERSTATE_FOGVERTEXMODE, Mode)
        Call g_d3dDevice.SetRenderStateSingle(D3DRENDERSTATE_FOGSTART, StartFog)
        Call g_d3dDevice.SetRenderStateSingle(D3DRENDERSTATE_FOGEND, EndFog)
    Else
        Call g_d3dDevice.SetRenderState(D3DRENDERSTATE_FOGVERTEXMODE, Mode)
        Call g_d3dDevice.SetRenderStateSingle(D3DRENDERSTATE_FOGDENSITY, Density)
    End If

    ' Enable range-based fog if desired (only supported for vertex fog).
    ' For this example, it is assumed that bUseRange 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 bUseRange = True Then
        Call g_d3dDevice.SetRenderState( _
            D3DRENDERSTATE_RANGEFOGENABLE, True)
    End If
End Sub
```

Geometry Blending

This section contains information about a new feature in DirectX 7.0: geometry blending. The following topics are discussed:

- About Geometry Blending
- Blending Transform and Render States
- Blending Weights
- Using Geometry Blending

About Geometry Blending

Direct3D Immediate Mode for DirectX 7.0 enables applications to increase the realism of their 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, then performing a linear blend between the resultant vertices to create a single set of geometry for rendering. The following image, based on the banana rendered by the Bend Sample, illustrates.

The preceding 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 different world matrices (including the standard world matrix, D3DTRANSFORMSTATE_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 D3DRENDERSTATE_NORMALIZENORMALS render state is set to TRUE.

Without geometry blending, dynamic articulated models are often rendered in segments. For instance, take 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 illustrates:

```
typedef struct _Arm {
```

```
D3DVERTEX upper_arm_verts[200];
D3DMATRIX matWorld_Upper;
```

```
D3DVERTEX lower_arm_verts[200];
D3DMATRIX matWorld_Lower;
} ARM, *LPARM;
```

ARM MyArm; // This will need to be initialized.

To render the arm, two rendering calls would be made, as shown in the following code:

```
// Render the upper arm.
d3dDevice->SetTransform( D3DTRANSFORMSTATE_WORLD, &MyArm.matWorld_Upper );
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, D3DFVF_VERTEX,
    MyArm.upper_arm_verts, 200, 0 );

// 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( D3DTRANSFORMSTATE_WORLD, &MyArm.matWorld_Lower );
d3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, D3DFVF_VERTEX,
    MyArm.lower_arm_verts, 200, 0 );
```

[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 D3DRENDERSTATE_NORMALIZENORMALS render state is set to True.

Without geometry blending, dynamic articulated models are often rendered in segments. For instance, take 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 pseudo-code illustrates:

```
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 will need to be initialized.
To render the arm, two rendering calls would be made, as shown in the following code:

```
' Render the upper arm.  
Call d3DDevice.SetTransform(D3DTRANSFORMSTATE_WORLD, MyArm.matWorld_Upper)  
Call d3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, D3DFVF_VERTEX, _  
    MyArm.upper_arm_verts, 200, 0)  
  
' 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(D3DTRANSFORMSTATE_WORLD, MyArm.matWorld_Lower)  
Call d3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, D3DFVF_VERTEX, _  
    MyArm.lower_arm_verts, 200, 0)
```

The following image is the banana from a version of the Bend Sample, modified to use this technique.

The differences between the blended geometry and the non-blended geometry are obvious. This example is admittedly somewhat extreme. In a real-world application, the "joints" of segmented models are designed such that seams aren't as obvious. However, ugly 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. Geometry blending isn't free; 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 **IDirect3DDevice7::SetTransform** and **IDirect3DDevice7::SetTransform** methods recognize the **D3DTRANSFORMSTATE_WORLD**, **D3DTRANSFORMSTATE_WORLD1**, **D3DTRANSFORMSTATE_WORLD2**, and **D3DTRANSFORMSTATE_WORLD3** members of the **D3DTRANSFORMSTATETYPE** enumerated type to identify the matrices between which geometry will be blended. (The **D3DTRANSFORMSTATE_WORLD** member might be thought of as "D3DTRANSFORMSTATE_WORLD0".)

The **D3DRENDERSTATETYPE** enumerated type includes the **D3DRENDERSTATE_VERTEXBLEND** render state to enable and control geometry blending. Valid values for this renderstate are defined by the **D3DVERTEXBLEND_FLAGS** enumerated type, which includes the **D3DVBLEND_DISABLE**, **D3DVBLEND_1WEIGHT**, **D3DVBLEND_2WEIGHT**, **D3DVBLEND_3WEIGHT** members. If geometry blending is enabled, the vertex format must include the appropriate number of blending weights.

[\[Visual Basic\]](#)

The **Direct3DDevice7.SetTransform** and **Direct3DDevice7.SetTransform** methods recognize the **D3DTRANSFORMSTATE_WORLD**, **D3DTRANSFORMSTATE_WORLD1**, **D3DTRANSFORMSTATE_WORLD2**, and **D3DTRANSFORMSTATE_WORLD3** members of the **CONST_D3DTRANSFORMSTATETYPE** enumeration to identify the matrices between which geometry will be blended. (The **D3DTRANSFORMSTATE_WORLD** member might be thought of as "D3DTRANSFORMSTATE_WORLD0".)

The **CONST_D3DRENDERSTATETYPE** enumeration includes the **D3DRENDERSTATE_VERTEXBLEND** render state to enable and control geometry blending. Valid values for this renderstate are defined by the **CONST_D3DVERTEXBLEND_FLAGS** enumerated type, which includes the **D3DVBLEND_DISABLE**, **D3DVBLEND_1WEIGHT**, **D3DVBLEND_2WEIGHT**, **D3DVBLEND_3WEIGHT** members. 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 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 Direct3D rendering methods.

The system performs a linear blend between the weighted results of the blend matrices. The following is the complete blending formula:

$$vBlend = V_1W_1 + \dots + V_{n-1}W_{n-1} + V_n \left(1.0 - \sum_{i=1}^n W_i \right)$$

In the preceding formula, *vBlend* is the output vertex, the *v*-elements are the vertices produced by the applied world matrix (D3DTRANSFORMSTATE_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 such that the sum of all weights is 1.0, expressed in sigma notation here. This formula can be simplified for each of the cases currently supported by Direct3D:

$$vBlend = V_1W_1 + V_2(1.0 - W_1)$$

$$vBlend = V_1W_1 + V_2W_2 + V_3(1.0 - (W_1 + W_2))$$

$$vBlend = V_1W_1 + V_2W_2 + V_3W_3 + V_4(1.0 - (W_1 + W_2 + W_3))$$

The preceding are the simplified forms of the complete blending formula for the two, three, and four blend matrix cases (the only cases currently supported).

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 DirectX.

Using Geometry Blending

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 could be used for vertices that will be blended between two matrices:

```
//
// The flexible vertex format descriptor for this vertex would be:
//
//   dwFVF = 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 D3DTRANSFORMSTATE_WORLD and D3DTRANSFORMSTATE_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.

[Visual Basic]

The following user-defined type could be used for vertices that will be blended between two matrices:

```
'
'
' The flexible vertex format descriptor for this vertex would be:
'
'   IFVF = (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

D3DTRANSFORMSTATE_WORLD and D3DTRANSFORMSTATE_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 will look 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

About Vertex Formats, Flexible Vertex Format Flags

Setting Blending Matrices

[C++]

You set the transformation matrices between which the system blends by calling the **IDirect3DDevice7::SetTransform** method. Set the first parameter to one of the "D3DTRANSFORMSTATE_WORLD" members from the **D3DTRANSFORMSTATETYPE** enumerated type, and set the second parameter to the address of the matrix to be set.

The following C++ code fragment sets two world matrices, between which geometry will be blended to create the illusion of a jointed arm:

```
// For this example, the pd3dDevice variable is assumed to be a
// valid pointer to an IDirect3DDevice7 interface for an initialized
// 3-D scene.
float    fBendAngle = 3.1415926f / 4.0f; // 45 degrees
D3DMATRIX matUpperArm, matLowerArm;

// The upper arm is immobile, use the identity matrix.
D3DUtil_SetIdentityMatrix( matUpperArm );
pd3dDevice->SetTransform( D3DTRANSFORMSTATE_WORLD, &matUpperArm );

// The lower arm rotates about the x-axis, attached to the upper arm.
D3DUtil_SetRotateXMatrix( matLowerArm, fBendAngle );
pd3dDevice->SetTransform( D3DTRANSFORMSTATE_WORLD1, &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 **Direct3DDevice7.SetTransform** method. Set the first parameter to one of the "D3DTRANSFORMSTATE_WORLD" members from the **CONST_D3DTRANSFORMSTATETYPE** enumeration, and set the second parameter to the address of the matrix to be set.

The following Visual Basic code fragment 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 Direct3DDevice7 object for an initialized
' 3-D scene. The dx variable is a valid reference to a DirectX7 object.
Dim nBendAngle As Single
Dim matUpperArm As D3DMATRIX, _
    matLowerArm As D3DMATRIX

nBendAngle = 3.1415926 / 4# ' 45 degrees

' The upper arm is immobile, use the identity matrix.
Call dx.IdentityMatrix(matUpperArm)
Call d3dDevice.SetTransform(D3DTRANSFORMSTATE_WORLD, matUpperArm)

' The lower arm rotates about the x-axis, attached to the upper arm.
Call dx.RotateXMatrix(matLowerArm, -nBendAngle)
Call d3dDevice.SetTransform(D3DTRANSFORMSTATE_WORLD1, 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.
```

Enabling Geometry Blending

[C++]

Geometry blending is disabled by default. To enable geometry blending, simply call the **IDirect3DDevice7::SetRenderState** method to set the **D3DRENDERSTATE_VERTEXBLEND** render state to a value from the **D3DVERTEXBLEND_FLAGS** enumerated type. The following code fragment shows what this call might look like when setting the render state for a blend between two world matrices.

```
pd3dDevice->SetRenderState( D3DRENDERSTATE_VERTEXBLEND,
    D3DVBLEND_1WEIGHT );
```

When **D3DRENDERSTATE_VERTEXBLEND** is set to any value other than **D3DVBLEND_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 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, simply call the **Direct3DDevice7.SetRenderState** method to set the **D3DRENDERSTATE_VERTEXBLEND** render state to a value from the **CONST_D3DVERTEXBLEND_FLAGS** enumeration. The following code fragment shows what this call might look like when setting the render state for a blend between two world matrices.

```
Call d3dDevice.SetRenderState(D3DRENDERSTATE_VERTEXBLEND,  
D3DVBLEND_1WEIGHT)
```

When **D3DRENDERSTATE_VERTEXBLEND** is set to any value other than **D3DVBLEND_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 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.

User-defined Clip Planes

Direct3D enables applications to clip geometry according to arbitrarily defined-clip planes. This following topics introduce user-defined clip planes, and provide information about how you can use them in your application.

- What Are User-defined Clip Planes?
- Using User-defined Clip Planes

What Are User-defined Clip Planes?

User-defined clip planes enable applications to clip geometry in a scene by applying arbitrary clip planes. These clipping planes are described by the general plane equation, which the system assumes to be in world-space.

$$Ax + By + Cz + D = 0$$

The following image is taken from a version of the Spheremap sample, slightly modified to emphasize the effects of clipping. (The sample was modified to render in wire-frame, with back-face culling disabled.) In this image, no clip planes are applied.

When user-defined clipping planes are applied, you can easily clip the geometry in a scene. A point with world-coordinates coordinates (x, y, z) are clipped from the half space of the plane if the following inequality is satisfied.

$$Ax + By + Cz + D \geq 0$$

In the preceding formula, A, B, C, and D are coefficients supplied by an application to the system to define a plane. Points that exist on or behind the clipping plane are clipped from the scene. The following image shows two teapots, clipped by two different planes.

The teapot on the left has been clipped by a plane that has the coefficients [A = -1.0, B = 0.0, C = 0.0, D = 0.0], which causes the system to clip all vertices with $x \geq 0$. The teapot on the right was clipped by a plane defined by the coefficients [A = 1.0, B = 0.0, C = 1.0, D = 0.0], which clips all vertices at $z \geq 0.0$.

You can clip geometry by clipping with multiple planes. When you use multiple clip planes, the rendered area is the union of the half-spaces defined by each plane. The following image shows the teapot clipped by two planes.

The teapot in the preceding image was clipped by two planes: one with the coefficients [A = 1.0, B = 0.0, C = 0.0, D = 0.0], and another plane defined by the [A = 0.0, B = -1.0, C = 0.0, D = 0.0] coefficients.

[C++]

Direct3D supports clipping with up to 32 (D3DMAXUSERCLIPPLANES) clipping planes, although device support can vary. To be sure, always check device capabilities for the current device (the **wMaxUserClipPlanes** member of the **D3DDEVICEDESC7** structure).

[Visual Basic]

Direct3D supports clipping with up to 32 clipping planes, although device support can vary. To be sure, always check device capabilities for the current device (the **nMaxUserClipPlanes** member of the **D3DDEVICEDESC7** type).

User-defined clip planes can be used for much more than the simple geometry clipping shown here, although this can be useful in many cases. You might combine

user-defined clip planes and stencil buffers to perform advanced tasks like capping, or interference testing.

Using User-defined Clip Planes

User-defined clip planes are simple to define, manipulate, and enable. The following topics provide information about these common tasks.

- Setting and Retrieving Clip Planes
- Enabling and Disabling Clip Planes

Setting and Retrieving Clip Planes

[C++]

You set and retrieve user-defined clip planes by calling the **IDirect3DDevice7::SetClipPlane** and **IDirect3DDevice7::GetClipPlane** methods. The coefficients that describe the plane are in a four-element array of type **D3DVALUE**.

The **SetClipPlane** method accepts two parameters. The first parameter is the zero-based index of the clip plane being set, and the second parameter is the address of a buffer that contains the coefficients for the general plane equation.

Note

The coefficients passed to **SetClipPlane** take the form of the general plane equation. If the values in the array you pass with the method were labeled A, B, C, and D (in the order they appear in memory), they would fit into the general plane equation so that $Ax + By + Cz + D = 0$. A point with world-space coordinates (x, y, z) is visible in the half space of the plane if $Ax + By + Cz + D \geq 0$. Points that exist on or behind the clipping plane are clipped from the scene.

The following code shows how this method is called in C++.

```
// For this example, the pd3dDevice variable is a valid
// pointer to an IDirect3DDevice7 interface.
```

```
// Set-up a clipping plane such that vertices with
// x<=0 will be clipped.
D3DVALUE clip[4];
ZeroMemory( (LPVOID)clip, sizeof(clip) );
```

```
clip[0] = 1.0f;
pd3dDevice->SetClipPlane( 0, clip );
```

Setting coefficients for a clip plane does not enable clipping with that plane. For more information, see Enabling and Disabling Clip Planes.

You can define up to 32 (D3DMAXUSERCLIPPLANES) clip planes. Therefore, the highest index you can pass to the **SetClipPlane** method is 31. If you pass an invalid value, the method will fail, returning **DDERR_INVALIDPARAMS**.

The **IDirect3DDevice7::GetClipPlane** method is semantically similar to its counterpart. The following C++ code retrieves the coefficients currently set for the clip plane at index 2.

```
// For this example, the pd3dDevice variable is a valid
// pointer to an IDirect3DDevice7 interface.

D3DVALUE clip[4];
pd3dDevice->GetClipPlane( 2, clip );
```

[Visual Basic]

You set and retrieve user-defined clip planes by calling the **Direct3DDevice7.SetClipPlane** and **Direct3DDevice7.GetClipPlane** methods. The coefficients that describe the plane are in a four-element array of type **D3DVALUE**.

The first parameter to **SetClipPlane** is the zero-based index of the clip plane being set, and the remaining parameters contain the coefficients for the general plane equation.

Note

The coefficients passed to **SetClipPlane** take the form of the general plane equation. If the values in the array you pass with the method were labeled A, B, C, and D (in the order they appear in memory), they would fit into the general plane equation so that $Ax + By + Cz + D = 0$. A point with world-space coordinates (x, y, z) is visible in the half space of the plane if $Ax + By + Cz + D \geq 0$. Points that exist on or behind the clipping plane are clipped from the scene.

The following code shows how this method is called in Visual Basic.

```
' For this example, the d3dDevice variable is a valid
' reference to an Direct3DDevice object.

' Set-up a clipping plane such that vertices with
' x<=0 will be clipped.
Dim A As Single, B As Single, _
    C As Single, D As Single

A = 1#

Call d3dDevice.SetClipPlane(0, A, B, C, D)
```

Setting coefficients for a clip plane does not enable clipping with that plane. For more information, see [Enabling and Disabling Clip Planes](#).

You can define up to 32 (D3DMAXUSERCLIPPLANES) clip planes. Therefore, the highest index you can pass to the **SetClipPlane** method is 31. If you pass an invalid value, the method will fail, returning DDERR_INVALIDPARAMS.

The **Direct3DDevice7.GetClipPlane** method is semantically similar to its counterpart. The following Visual Basic code retrieves the coefficients currently set for the clip plane at index 2.

' For this example, the d3dDevice variable is a valid
' reference to an Direct3DDevice object.

```
Dim A As Single, B As Single, _  
    C As Single, D As Single
```

```
Call d3dDevice.GetClipPlane(2, A, B, C, D)
```

Enabling and Disabling Clip Planes

[C++]

It's important to remember that clip planes are not automatically enabled when you call the **IDirect3DDevice7::SetClipPlane** method. Use the **D3DRENDERSTATE_CLIPPLANEENABLE** render state to enable or disable the clip planes currently set for the device. This render state accepts a **DWORD** value, where each bit starting with the least significant bit corresponds to a clip plane. If a bit is set, the clip plane is enabled; if the bit is cleared, the corresponding plane is disabled. In the value, bit 0 enables and disables clip plane at index zero, bit 1 controls clip plane one, and so on to bit 31. (Direct3D can clip with up to 32 planes at once, but device support can vary.) These semantics allow applications to enable or disable every clip plane in a single call.

The **d3dtypes.h** header file defines the **D3DCLIPPLANEn** macros to create patterns that you can combine and pass as values for the **D3DRENDERSTATE_CLIPPLANEENABLE** render state. The following C++ code uses this macro.

```
// For this example, the pd3dDevice variable is a valid  
// pointer to an IDirect3DDevice7 interface.  
  
// Enable clip plane 0 and 1, while implicitly disabling  
// all other user-defined clip planes.  
pd3dDevice->SetRenderState( D3DRENDERSTATE_CLIPPLANEENABLE,  
    D3DCLIPPLANE0 | D3DCLIPPLANE1);
```

[Visual Basic]

It's important to remember that clip planes are not automatically enabled when you call the **Direct3DDevice7.SetClipPlane** method. Use the **D3DRENDERSTATE_CLIPPLANEENABLE** render state to enable or disable the clip planes currently set for the device. This render state values from the **CONST_D3DCLIPPLANEFLAGS** enumeration. Combine values from this enumeration to enable the corresponding clip plane, or omit them to disable the

corresponding clip plane. (Use the D3DCPF_DISABLEALL value to disable all clip planes.) These semantics allow applications to enable or disable every clip plane in a single call.

The following Visual Basic code shows how this method is used.

```
' For this example, the d3dDevice variable is a valid  
' reference to a Direct3DDevice7 object.  
  
' Enable clip plane 0 and 1, while implicitly disabling  
' all other user-defined clip planes.  
Call d3dDevice.SetRenderState(D3DRENDERSTATE_CLIPPLANEENABLE, _  
                             (D3DCPF_ENABLEPLANE0 Or D3DCPF_ENABLEPLANE1))
```

Environment Mapping

This section provides information about performing two common types of environment mapping with Direct3D Immediate Mode. 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.

The following image is taken from the Spheremap sample (C++) which, as its name implies, uses a type of environment mapping called "spherical environment mapping." For details, see Spherical Environment Mapping.

The teapot in the preceding image appears to reflect its surroundings, 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.

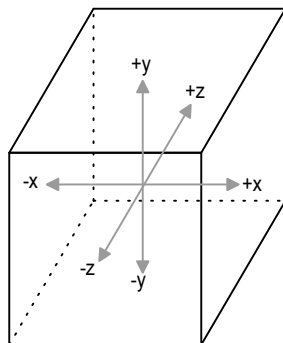
Cubic Environment Mapping

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:

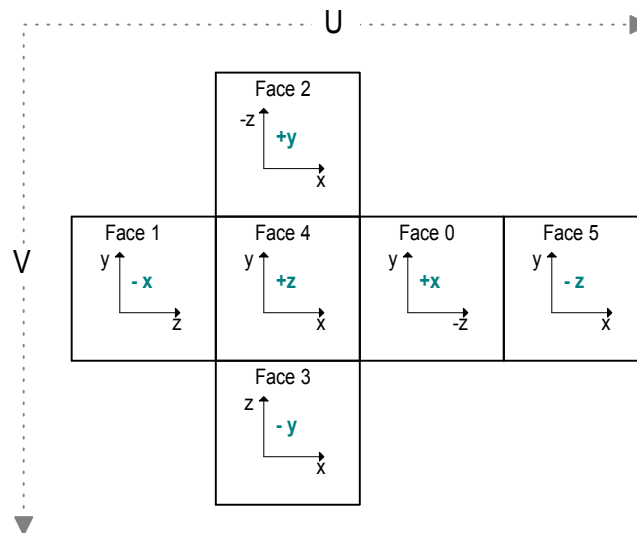
- What Are Cubic Environment Maps?
- Creating Cubic Environment Map Surfaces
- Accessing Cubic Environment Map Faces
- Mipmapped Cubic Environment Maps
- Rendering to Cubic Environment Maps
- Texture Coordinates for Cubic Environment Maps

What Are Cubic Environment Maps?

A 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 ninety 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 attached texture surfaces, created in a single call, which can be accessed like any other set of attached surfaces. 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 DDSCAPS_3DDEVICE capability.)

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

You create a cubic environment map by calling the **IDirectDraw7::CreateSurface** method in C++ or **DirectDraw7.CreateSurface** in Visual Basic. Cube maps are complex surfaces; that is, they are a set of attached surfaces, created by DirectDraw in a single call. Surfaces that were created individually cannot be attached to each other to create a cubic-environment map. Cubic-environment map textures must be square, with dimensions that are a powers-of-two.

[C++]

The following code shows how your C++ application might create a simple cubic-environment map.

```
// For this example, the pDD variable is a valid pointer
// to an IDirectDraw7 interface.
DDSURFACEDESC2 ddsd;
```

```

ZeroMemory((LPVOID)&ddsd, sizeof(DDSURFACEDESC2));

ddsd.dwSize      = sizeof(DDSURFACEDESC2);
ddsd.dwFlags     = DDSD_CAPS | DDSD_WIDTH |
                  DDSD_HEIGHT | DDSD_PIXELFORMAT;
ddsd.ddsCaps.dwCaps = DDSCAPS_TEXTURE;

// Set the pixel format to a valid texture format here.

// Dimensions are an arbitrary, but must be a power-of-two.
ddsd.dwWidth  = 64;
ddsd.dwHeight = 64;

// Set caps for a system memory cube-map texture that is a valid
// render-target surface.
ddsd.ddsCaps.dwCaps = DDSCAPS_COMPLEX | DDSCAPS_3DDEVICE |
                    DDSCAPS_TEXTURE;
ddsd.ddsCaps.dwCaps2 = DDSCAPS2_CUBEMAP | DDSCAPS2_CUBEMAP_ALLFACES;

LPDIRECTDRAW7 pddsCubeMap;

// Create the cube map.
if( FAILED( pDD->CreateSurface( &ddsd, pddsCubeMap, NULL ) ) )
{
    // code to handle error goes here.
}

```

Note

If you intend to render into the faces of a cube map, the surface description you provide when you create the surface must contain the DDSCAPS_3DDEVICE capability.

Cube maps can be managed textures. If you include either the DDSCAPS2_TEXTUREMANAGE or DDSCAPS2_TEXTUREMANAGE capability flag in the surface capabilities when you create the surface, the resulting texture will be managed. For more information, see Automatic Texture Management.

[\[Visual Basic\]](#)

The following Visual Basic code shows how your application might create a simple cubic-environment map.

```

' For this example, the dd variable is a valid reference
' to a DirectDraw7 object.
On Local Error Resume Next

```

```

Dim ddsd As DDSURFACEDESC2

```

```
ddsd.IFlags = DDSD_CAPS Or DDSD_WIDTH Or _  
              DDSD_HEIGHT Or DDSD_PIXELFORMAT  
  
ddsd.ddsCaps.dwCaps = DDSCAPS_TEXTURE  
  
' Set the pixel format to a valid texture format here.  
  
' Dimensions are an arbitrary, but must be a power-of-two.  
ddsd.dwWidth = 64: ddsd.dwHeight = 64  
  
' Set caps for a system memory cube-map texture that is a valid  
' render-target surface.  
ddsd.ddsCaps.dwCaps = DDSCAPS_COMPLEX Or DDSCAPS_3DDEVICE Or _  
                      DDSCAPS_TEXTURE  
ddsd.ddsCaps.dwCaps2 = DDSCAPS2_CUBEMAP Or DDSCAPS2_CUBEMAP_ALLFACES  
  
Dim ddsCubeMap As DirectDrawSurface7  
  
' Create the cube map.  
Set ddsCubeMap = dd.CreateSurface(ddsd)  
  
If Err.Number <> DD_OK Then  
    ' code to handle error goes here.  
End If
```

Note

If you intend to render into the faces of a cube map, the surface description you provide when you create the surface must contain the DDSCAPS_3DDEVICE capability.

Cube maps can be managed textures. If you include either the DDSCAPS2_TEXTUREMANAGE or DDSCAPS2_TEXTUREMANAGE capability flag in the surface capabilities when you create the surface, the resulting texture will be managed. For more information, see Automatic Texture Management.

Accessing Cubic Environment Map Faces

[C++]

Like any other complex surface, you navigate between faces of a cubic environment map by using the **IDirectDrawSurface7::GetAttachedSurface** method. The **GetAttachedSurface** method accepts a pointer to a **DDSCAPS2** structure that describes the attached surface the method should retrieve. Set the **dwCaps2** member to any of the flag values beginning with "DDSCAPS2_CUBEMAP_", combined with the DDSCAPS2_CUBEMAP flag, to retrieve the corresponding face.

The following code retrieves the cube-map surface used for the positive-y face (face 2).

```
// For this example, the pddsMainFace variable contains a valid
// pointer to the face returned by CreateSurface when the cube map
// was originally created.
LPDIRECTDRAW7 pddsFace2;

DDSCAPS2 ddsCaps;
ddsCaps.dwCaps2 = DDSCAPS2_CUBEMAP_POSITIVEY | DDSCAPS2_CUBEMAP;

HRESULT hr;
hr = pddsMainFace->GetAttachedSurface( &ddsCaps,
                                       &pddsFace2);

if( FAILED(hr) )
{
    // code to handle error goes here.
}
```

[Visual Basic]

Like any other complex surface, you navigate between faces of a cubic environment map by using the **DirectDrawSurface7.GetAttachedSurface** method. The **GetAttachedSurface** method accepts a **DDSCAPS2** type that describes the attached surface the method should retrieve. Set the **ICaps2** member to any of the flag values from the **CONST_DDSSURFACECAPS2FLAGS** enumeration that begin with "DDSCAPS2_CUBEMAP_", combined with the DDSCAPS2_CUBEMAP flag, to retrieve the corresponding face.

The following code retrieves the cube-map surface used for the positive-y face (face 2).

```
' For this example, the ddsMainFace variable contains a valid
' reference to the face returned by CreateSurface when the cube map
' was originally created.
Dim ddsFace2 As DirectDrawSurface7
Dim ddsCaps As DDSCAPS2

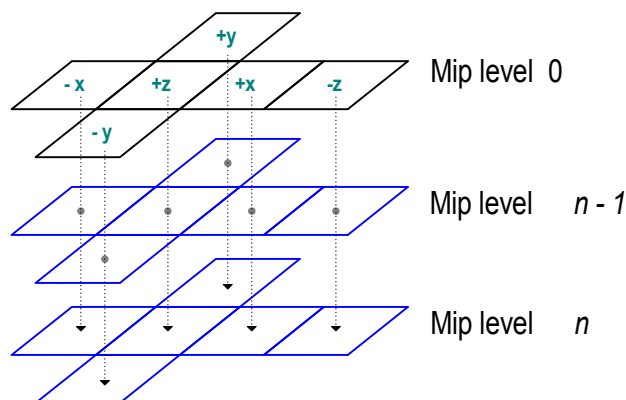
ddsCaps.ICaps2 = DDSCAPS2_CUBEMAP_POSITIVEY Or DDSCAPS2_CUBEMAP

Set ddsFace2 = ddsMainFace.GetAttachedSurface(ddsCaps)

If Err.Number <> DD_OK Then
    ' code to handle error goes here.
End If
```

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 might envision the topography of these surfaces as follows:



[C++]

Applications that create mipmapped cubic-environment maps can access each face by calling the **IDirectDrawSurface7::GetAttachedSurface** method, passing a one of the flags that begins with "DDSCAPS2_CUBEMAP_" flag to access the corresponding face (as discussed in Accessing Cubic Environment Map Faces). Then, the subordinate mipmap levels can be retrieved by calling the **IDirectDrawSurface7::GetAttachedSurface** for that face, passing the DDSCAPS_MIPMAP flag.

[Visual Basic]

Applications that create mipmapped cubic-environment maps can access each face by calling the **DirectDrawSurface7.GetAttachedSurface** method, passing a one of the flags from the **CONST_DDSSURFACECAPS2FLAGS** enumeration that begins with "DDSCAPS2_CUBEMAP_" to access the corresponding face (as discussed in Accessing Cubic Environment Map Faces). Then, the subordinate mipmap levels can be retrieved by calling the **DirectDrawSurface7.GetAttachedSurface** for that face, passing the DDSCAPS_MIPMAP flag.

Rendering to Cubic Environment Maps

If you create a cubic environment map with the DDSCAPS_3DDEVICE capability, you can render to the individual faces of the cube map just like you would any other render-target surface. The most important thing to do before rendering to a face is set the transformation matrices such 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 prepares and sets a view matrix according to the face being rendered.

```
//
// The ppddsFaces variable is the address of an array of IDirectDrawSurface7
// interface pointers, one for each face.
//
void RenderFaces(LPDIRECTDRAWSURFACE7* ppddsFaces)
{
    // Loop through the six faces of the cube map.
    for( DWORD i=0; i<6; i++ )
    {
        DDSCAPS2 ddsc;
        ZeroMemory( (LPVOID)&ddsc, sizeof(DDSCAPS2) );

        // Get the surface caps (so we can find out which face this is).
        ppddsFaces[i]->GetCaps( &ddsc );

        // Standard view that will be overridden below
        D3DVECTOR vEnvEyePt = D3DVECTOR( 0.0f, 0.0f, 0.0f );
        D3DVECTOR vLookatPt, vUpVec;

        switch( ddsc.dwCaps2 & DDSCAPS2_CUBEMAP_ALLFACES )
        {
            case DDSCAPS2_CUBEMAP_POSITIVEX:
                vLookatPt = D3DVECTOR( 1.0f, 0.0f, 0.0f );
                vUpVec   = D3DVECTOR( 0.0f, 1.0f, 0.0f );
                break;
            case DDSCAPS2_CUBEMAP_NEGATIVEX:
                vLookatPt = D3DVECTOR( -1.0f, 0.0f, 0.0f );
                vUpVec   = D3DVECTOR( 0.0f, 1.0f, 0.0f );
                break;
            case DDSCAPS2_CUBEMAP_POSITIVEY:
                vLookatPt = D3DVECTOR( 0.0f, 1.0f, 0.0f );
                vUpVec   = D3DVECTOR( 0.0f, 0.0f, -1.0f );
                break;
            case DDSCAPS2_CUBEMAP_NEGATIVEY:
                vLookatPt = D3DVECTOR( 0.0f, -1.0f, 0.0f );
                vUpVec   = D3DVECTOR( 0.0f, 0.0f, 1.0f );
                break;
            case DDSCAPS2_CUBEMAP_POSITIVEZ:
                vLookatPt = D3DVECTOR( 0.0f, 0.0f, 1.0f );
                vUpVec   = D3DVECTOR( 0.0f, 1.0f, 0.0f );
```

```

        break;
    case DDSCAPS2_CUBEMAP_NEGATIVEZ:
        vLookatPt = D3DVECTOR( 0.0f, 0.0f,-1.0f );
        vUpVec    = D3DVECTOR( 0.0f, 1.0f, 0.0f );
        break;
    }

    D3DMATRIX matView;
    D3DUtil_SetViewMatrix( matView, vEnvEyePt, vLookatPt, vUpVec );
    g_pd3dDevice->SetTransform( D3DTRANSFORMSTATE_VIEW, &matView );

```

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, perhaps), take care to set the projection matrix accordingly.

This code creates and sets a projection matrix for the most common case.

```

    // Use 90-degree field of view in the projection.
    D3DMATRIX matProj;
    D3DUtil_SetProjectionMatrix( matProj, g_PI/2, 1.0f, 0.5f, 1000.0f );
    g_pd3dDevice->SetTransform( D3DTRANSFORMSTATE_PROJECTION, &matProj );

```

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, provided for completeness, outlines this task.

```

//
// Render the scene.
//

// Swap the depth-buffer to a face, then sets it as the render target.
// This function is described in more detail later.
ChangeRenderTarget( ppddsFaces[i] );

// Clear the zbuffer.
g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_ZBUFFER, 0x000000ff, 1.0f, 0L );

// Enable antialiasing, it makes the environment maps look better.
g_pd3dDevice->SetRenderState( D3DRENDERSTATE_ANTIALIAS,
                             D3DANTIALIAS_TRANSLUCENTSORTINDEPENDENT );

// Begin the scene.
if( SUCCEEDED( g_pd3dDevice->BeginScene() ) )
{
    // Position the geometry of the scene, as normal, by setting the
    // world matrix for each object to be rendered.

```

```

        g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST,
                                     D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1 |
D3DFVF_TEXCOORDSIZE3(0),
                                     lpVertices, VERTEX_COUNT, 0);

        // End the scene.
        g_pd3dDevice->EndScene();
    }
}
}

```

Note

Most applications should enable antialiasing when rendering to the faces of a cubic environment map. Because each face is relatively small, enabling antialiasing when rendering to them can noticeably improve the appearance of the resulting environment map.

Notice the call to the `ChangeRenderTarget` function. When rendering to the cube map faces, you must of course 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 following code uses the latter method.

```

HRESULT ChangeRenderTarget( LPDIRECTDRAW_SURFACE7 pddsNewRenderTarget)
{
    LPDIRECTDRAW_SURFACE7 pddsOldRenderTarget = NULL;
    g_pd3dDevice->GetRenderTarget( &pddsOldRenderTarget );

    if( pddsOldRenderTarget )
    {
        LPDIRECTDRAW_SURFACE7 pddsZBuffer = NULL;
        DDSCAPS2 ddscaps = { DDSCAPS_ZBUFFER, 0, 0, 0 };
        pddsOldRenderTarget->GetAttachedSurface( &ddscaps, &pddsZBuffer );

        if( pddsZBuffer )
        {
            pddsOldRenderTarget->DeleteAttachedSurface( 0, pddsZBuffer );
            pddsNewRenderTarget->AddAttachedSurface( pddsZBuffer );
            pddsZBuffer->Release();
        }
        pddsOldRenderTarget->Release();
    }

    g_pd3dDevice->SetRenderTarget( pddsNewRenderTarget, 0 );
    return S_OK;
}

```

[\[Visual Basic\]](#)

The following Visual Basic code prepares and sets a view matrix according to the face being rendered.

```
'
' The ddsFaces variable is the address of an array of DirectDrawSurface7
' objects, one for each face.
'

Sub RenderFaces(ddsFaces() As DirectDrawSurface7)
    Dim i As Integer

    ' Loop through the six faces of the cube map.
    For i = 0 To UBound(ddsFaces)
        Dim ddsc As DDSCAPS2
        Dim vEnvEyePt As D3DVECTOR, _
            vLookatPt As D3DVECTOR, _
            vUpVec As D3DVECTOR

        ' Get the surface caps (so we can find out which face this is).
        Call ddsFaces(i).GetCaps(ddsc)

        Select Case (ddsc.dwCaps2 And DDSCAPS2_CUBEMAP_ALLFACES)
            Case DDSCAPS2_CUBEMAP_POSITIVEX:
                vLookatPt.x = 1#: vUpVec.y = 1#

            Case DDSCAPS2_CUBEMAP_NEGATIVEX:
                vLookatPt.x = -1#: vUpVec.y = 1#

            Case DDSCAPS2_CUBEMAP_POSITIVEY:
                vLookatPt.y = 1#: vUpVec.z = -1#

            Case DDSCAPS2_CUBEMAP_NEGATIVEY:
                vLookatPt.y = -1#: vUpVec.z = 1#

            Case DDSCAPS2_CUBEMAP_POSITIVEZ:
                vLookatPt.z = 1#: vUpVec.y = 1#

            Case DDSCAPS2_CUBEMAP_NEGATIVEZ:
                vLookatPt.z = -1#: vUpVec.y = 1#

        End Select

        Dim matView As D3DMATRIX
        Call g_dx.ViewMatrix(matView, vEnvEyePt, vLookatPt, vUpVec)
        Call g_pd3dDevice.SetTransform(D3DTRANSFORMSTATE_VIEW, matView)
    Next i
End Sub
```

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, perhaps), take care to set the projection matrix accordingly.

This code creates and sets a projection matrix for the most common case.

```
' Use 90-degree field of view in the projection.
Dim matProj As D3DMATRIX
Call g_dx.ProjectionMatrix(matProj, 0.5, 1000#, 3.1415926535 / 2)
Call g_d3dDevice.SetTransform(D3DTRANSFORMSTATE_PROJECTION, matProj)
```

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, provided for completeness, outlines this task.

```
'
' Render the scene.
'

' Swap the depth-buffer to a face, then sets it as the render target.
' This function is described in more detail later.
Call ChangeRenderTarget(ddsFaces(i))

Dim rec(1) As RECT
rec.Top = 0: rec.Left = 0

' This assumes surface dimensions of 64 by 64
rec.Bottom = 64: rec.Right = 64

' Clear the zbuffer.
Call g_d3dDevice.Clear(0, rec(), D3DCLEAR_ZBUFFER, _
    g_dx.CreateColorRGBA(0, 0, 256, 0), 1#, 0)

' Enable antialiasing, it makes the environment maps look better.
Call g_d3dDevice.SetRenderState(D3DRENDERSTATE_ANTIALIAS, _
    D3DANTIALIAS_TRANSLUCENTSORTINDEPENDENT)

' Begin the scene.
Call g_d3dDevice.BeginScene
If Err.Number = DD_OK Then
    ' Position the geometry of the scene, as normal, by setting the
    ' world matrix for each object to be rendered.

    Call g_pd3dDevice.DrawPrimitive(D3DPT_TRIANGLELIST, D3DFVF_XYZ Or
D3DFVF_NORMAL Or _
        D3DFVF_TEX1 Or D3DFVF_TEXCOORDSIZE3(0), _
        g_Vertices(0), VERTEX_COUNT, 0)
```

```
        ' End the scene.  
        Call g_d3dDevice.EndScene  
    End If  
Next i  
End Sub
```

Note

Most applications should enable antialiasing when rendering to the faces of a cubic environment map. Because each face is relatively small, enabling antialiasing when rendering to them can noticeably improve the appearance of the resulting environment map.

Notice the call to the `ChangeRenderTarget` subroutine. When rendering to the cube map faces, you must of course 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 following code uses the latter method.

```
Sub ChangeRenderTarget(ddsNewTarget As DirectDrawSurface7)  
    Dim ddsOldRenderTarget As DirectDrawSurface7  
  
    Set ddsOldRenderTarget = g_pd3dDevice.GetRenderTarget  
  
    If (ddsOldRenderTarget Is Not Nothing) Then  
        Dim ddsZBuffer As DirectDrawSurface7  
        Dim ddscaps As DDSCAPS2  
  
        ddscaps.ICaps = DDSCAPS_ZBUFFER  
        Set ddsZBuffer = ddsOldRenderTarget.GetAttachedSurface  
  
        If (ddsZBuffer Is Not Nothing) Then  
            Call ddsOldRenderTarget.DeleteAttachedSurface(ddsZBuffer)  
            Call ddsNewRenderTarget.AddAttachedSurface(ddsZBuffer)  
        End If  
    End If  
  
    Call g_pd3dDevice.SetRenderTarget(ddsNewRenderTarget)  
End Sub
```

Texture Coordinates for Cubic Environment Maps

[C++]

Texture coordinates that index a cubic-environment map aren't the simple *u, v* style coordinates used when standard textures are applied. In fact, cubic environment maps don't really use texture coordinates at all. In the 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. 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 would include the bit pattern created by the D3DFVF_TEXCOORDSIZE3 macro. The following code 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 will use the camera-space vertex normal for the vector. In other cases, like specular environment mapping, you would 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 a vector from the viewpoint to that vertex. Direct3D can automatically compute the reflection vectors for your geometry. Using this feature saves memory (you don't need to include texture coordinates for the environment map); it reduces bandwidth; and in the case of a TnLHAL Device, it can be significantly faster than the computations your application could make on its own. To take advantage of this feature, in the texture stage that contains the cubic-environment map, simply 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 shows how this value is used.

```
// The pd3dDevice variable is a valid pointer to an
// IDirect3DDevice7 interface.

// Automatically generate texture coordinates for stage 2. T
// This assumes that stage 2 is assigned a cube map.
// (Use the wrap mode from the texture coord set at index 1.)
pd3dDevice->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

D3DRENDERSTATE_LOCALVIEWER render state is set to TRUE, the formula used is the following:

$$R = 2(E \cdot N)N - E$$

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 D3DRENDERSTATE_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 D3DRENDERSTATE_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 the simple u, v style coordinates used when standard textures are applied. In fact, cubic environment maps don't really use texture coordinates at all. In the 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. Direct3D offers the **D3DFVF_TEXCOORDSIZEn** set of helper functions in the Math.bas source file for this purpose. (The Math.bas file ships with this SDK.) These 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 would include the bit pattern created by the **D3DFVF_TEXCOORDSIZE3** helper-function. The following code shows how this function is used.

```
' Create a flexible vertex format descriptor for a vertex that
' contains a position, normal, and one set of 3-D texture
' coordinates.
Dim IFVF As Long
IFVF = D3DFVF_XYZ Or D3DFVF_NORMAL Or
      D3DFVF_TEX1 Or D3DFVF_TEXCOORDSIZE3(0)
```

In some cases, such as diffuse light mapping, you will use the camera-space vertex normal for the vector. In other cases, like specular environment mapping, you would

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 a vector from the viewpoint to that vertex. Direct3D can automatically compute the reflection vectors for your geometry. Using this feature saves memory (you don't need to include texture coordinates for the environment map); it reduces bandwidth; and in the case of a TnLHAL Device, it can be significantly faster than the computations your application could make on its own. To take advantage of this feature, in the texture stage that contains the cubic-environment map, simply 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 shows how this value is used.

```
' The pd3dDevice variable is a valid reference to a
' Direct3DDevice7 object.

' Automatically generate texture coordinates for stage 2. T
' This assumes that stage 2 is assigned a cube map.
' (Use the wrap mode from the texture coord set at index 1.)
Call 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 D3DRENDERSTATE_LOCALVIEWER render state is set to True, the formula used is the following:

$$R = 2(E \cdot N)N - E$$

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 D3DRENDERSTATE_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 `D3DRENDERSTATE_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 into 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 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.

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 is the sphere map in use by the [Spheremap](#) sample (C++).

Texture Coordinates for Spherical Environment Maps

The texture coordinates 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 very 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.

$$u = \frac{N_x}{2} + 0.5$$
$$v = \frac{N_y}{2} + 0.5$$

In the preceding 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 preceding 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 of course 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. Note that this formula doesn't take account for the z component of the normal, but applications that use the formula might optimize their 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 will be culled when the object is rendered.

Applying Spherical Environment Maps

[C++]

You apply an environment map to objects in the same manner you would any other texture, by setting the texture into the appropriate texture stage with the **IDirect3DDevice7::SetTexture** method. Set the first parameter to the index for the desired texture stage, and set the second parameter to the address of the **IDirectDrawSurface7** interface returned when you created the 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 you would any other texture, by setting the texture into the appropriate texture stage with the **Direct3DDevice7.SetTexture** method. Set the first parameter to the index for the desired texture stage, and set the second parameter to the **DirectDrawSurface7** object returned when you created the 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.

Bump Mapping

This section provides information about performing bump mapping with Direct3D. The following topics introduce the 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?

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.

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 applies a bump map. The polygon count for the second globe is unchanged from that of the first globe.

Bump mapping in 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.

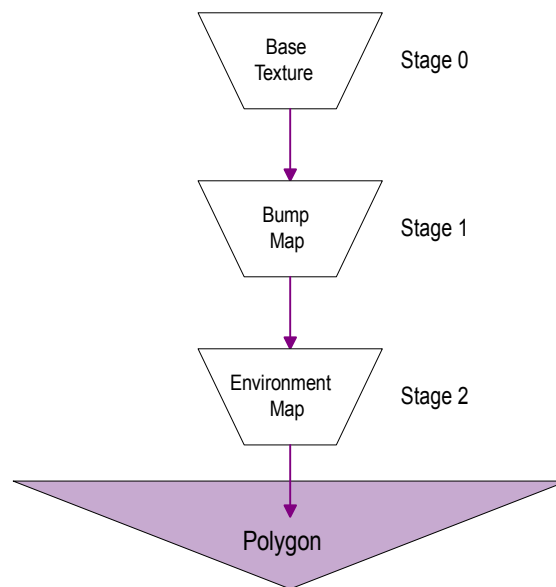
The BumpEarth sample uses a height-map to store contour data. When the sample starts, it processes the pixels in the height map to compute the appropriate u and v delta values (based on the relative height of each pixel to the four neighboring pixels).

Note

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.

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 DirectDraw surface 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++]

Like any other surface, you specify a bump map pixel format by using a **DDPIXELFORMAT** structure. The **DDPF_BUMPDUDV** flag in the **dwFlags** member of the structure identifies a pixel format for a bump map texture. When the flag is present, the **dwBumpBitCount**, **dwBumpDuBitMask**, **dwBumpDvBitMask**, and **dwBumpLuminanceBitMask** members can be used to describe the bit depth for the bump map, and the mask values for each pixel component.

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

You should enumerate supported bump map pixel formats by calling the **IDirect3DDevice7::EnumTextureFormats** method. For more information, see [Detecting Support for Bump Mapping](#).

[Visual Basic]

Like any other surface, you specify a bump map pixel format by using a **DDPIXELFORMAT** type. The **DDPF_BUMPDUDV** flag in the **IFlags** member of the type identifies a pixel format for a bump map texture. When the flag is present, the **IBumpBitCount**, **IBumpDuBitMask**, **IBumpDvBitMask**, and **IBumpLuminanceBitMask** members can be used to describe the bit depth for the bump map, and the mask values for each pixel component.

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

You should enumerate supported bump map pixel formats to ensure that the pixel format you intend to use is valid. Call the **Direct3DDevice7.GetTextureFormatsEnum** method to retrieve a reference to a **Direct3DEnumPixelFormats** enumerator class for texture formats. Use the enumerator to retrieve information about the supported texture formats, including bump map formats. For more information, see *Detecting Support for Bump Mapping*.

Bump Mapping Formulas

[C++]

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 the preceding 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 through 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 it uses to modulate the color of the environment map in the next blending stage.

$$L' = LS + O$$

In the preceding 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]

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 the preceding 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 it uses to modulate the color of the environment map in the next blending stage.

$$L' = LS + O$$

In the preceding 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 in this section provide details about the most common tasks your application will need to perform when using Direct3D bump mapping.

- Detecting Support For Bump Mapping
- Creating a Bump Map Texture
- Configuring Bump Mapping Parameters

Detecting Support For Bump Mapping

[C++]

A Direct3D device can perform bump mapping if it supports either of the D3DTOP_BUMPENVMAP or D3DTOP_BUMPENVMAPLUMINANCE texture blending operations. Additionally, applications should check the device capabilities to make sure it supports an appropriate number of blending stages (usually at least 3) 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.

```

BOOL SupportsBumpMapping()
{
    DDPIXELFORMAT ddpfBumpMap;
    D3DDEVICEDESC7 d3dDevDesc;
    ZeroMemory( &d3dDevDesc, sizeof(d3dDevDesc) );

    // Get the device capabilities.
    g_pd3dDevice->GetCaps( &d3dDevDesc );

    // Does this device support the two bump mapping blend operations?
    DWORD dwBumpOps = d3dDevDesc.dwTextureOpCaps &
        (D3DTEXOPCAPS_BUMPENVMAP |
        D3DTEXOPCAPS_BUMPENVMAPLUMINANCE);
    if ( 0 == dwBumpOps )
        return FALSE;

    // Does this device support up to three blending stages?
    if( d3dDevDesc.wMaxTextureBlendStages < 3 )
        return FALSE;

    //
    // Check for valid bump map pixel formats.
    //

    // The g_bFoundBumpFormat global variable will be set to TRUE
    // by the callback function if a valid format is found.

```

```

g_bFoundBumpFormat = FALSE;

g_pd3dDevice->EnumTextureFormats(TextureCallback, (LPVOID) &ddpfBumpMap);
if( FALSE == g_bFoundBumpFormat )
    return FALSE;

// The pixel format now in ddpfBumpMap can be used to create
// a surface format that's guaranteed to be valid.
return TRUE;
}

```

The following is the code for the callback function passes to the **IDirect3DDevice7::EnumTextureFormats** method.

```

HRESULT CALLBACK TextureCallback( DDPIXELFORMAT* pddpf, VOID* pddpfOut)
{
    // Take the first enumerated DuDv format.
    if( DDPF_BUMPDUDV == pddpf->dwFlags ){
        // Copy the format into the variable at pddpfOut
        // for use when creating the surface later.
        memcpy( pddpfOut, (LPVOID)pddpf, sizeof(DDPIXELFORMAT) );

        // Set the global flag to signal success.
        g_bFoundBumpFormat = TRUE;
        return D3DENUMRET_CANCEL;
    }

    return D3DENUMRET_OK;
}

```

[\[Visual Basic\]](#)

A Direct3D device can perform bump mapping if it supports either of the D3DTOP_BUMPENVMAP or D3DTOP_BUMPENVMAPLUMINANCE texture blending operations. Additionally, applications should check the device capabilities to make sure it supports an appropriate number of blending stages (usually at least 3) 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 Boolean
    Dim ddpfBumpMap As DDPIXELFORMAT
    Dim d3dDevDesc As D3DDEVICEDESC7

    SupportsBumpMapping = True

    ' Get the device capabilities.

```

```
Call g_d3dDevice.GetCaps(d3dDevDesc)

' Does this device support the two bump mapping blend operations?
Dim IBumpOps As Long
IBumpOps = (d3dDevDesc.ITextureOpCaps And _
            (D3DTEXOPCAPS_BUMPENVMAP Or
             D3DTEXOPCAPS_BUMPENVMAPLUMINANCE))

If IBumpOps = 0 Then SupportsBumpMapping = False

' Does this device support up to three blending stages?
If d3dDevDesc.nMaxTextureBlendStages < 3 Then SupportsBumpMapping = False

'
' Check for valid bump map pixel formats.
'

Dim BumpEnum As Direct3DEnumPixelFormatFormats
Set BumpEnum = g_d3dDevice.GetTextureFormatsEnum

Dim iLoop As Integer
For iLoop = 0 To BumpEnum.GetCount
    Call BumpEnum.GetItem(iLoop, ddpfBumpMap)

    ' Stop enumerating when a format is found.
    If ddpfBumpMap.IFlags And DDPF_BUMPDUDV Then Exit For
Next iLoop

If Not (ddpfBumpMap.IFlags And DDPF_BUMPDUDV) Then
    SupportsBumpMapping = False
End If
End Function
```

Creating a Bump Map Texture

You create a bump map texture like any other texture. Once your application has verified support for bump mapping and retrieved a valid pixel format (as discussed in [Detecting Support For Bump Mapping](#)), you can create a bump map texture surface. For information on creating surfaces, see [Obtaining a Texture Surface Object and Creating Surfaces](#).

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++]

0 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 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.
g_pd3dDevice->SetTexture( 0, lpd3dBaseTexture);
g_pd3dDevice->SetTexture( 1, lpd3dBumpMap);
g_pd3dDevice->SetTexture( 2, lpd3dEnvMap);
```

After setting the textures to their blending stages, the following code 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
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_SELECTARG1 );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE );
g_pd3dDevice->SetTextureStageState( 0, D3DTSS_TEXCOORDINDEX, 1 );

// Stage 1: the bump map
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_TEXCOORDINDEX, 1 );
// Use luminance for this example.
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP,
D3DTOP_BUMPENVMAPLUMINANCE);
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLORARG2, D3DTA_CURRENT );

// Stage 2: a specular environment map
```

```

g_pd3dDevice->SetTextureStageState( 2, D3DTSS_TEXCOORDINDEX, 0 );
g_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLOROP, D3DTOP_ADD );
g_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG1, D3DTA_TEXTURE );
g_pd3dDevice->SetTextureStageState( 2, D3DTSS_COLORARG2, D3DTA_CURRENT );

```

Once the blending operations and arguments are set, the following code 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); }
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT00, F2DW(1.0f) );
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT01, F2DW(0.0f) );
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVMAT10, F2DW(0.0f) );
g_pd3dDevice->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); }
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLSCALE, F2DW(0.5f) );
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_BUMPENVLOFFSET, F2DW(0.0f) );

```

After your application configures bump mapping parameters, it can render as it normally would, and the rendered polygons will receive bump mapping effects.

[Visual Basic]

0 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 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.
Call g_d3dDevice.SetTexture(0, d3dBaseTexture)
Call g_d3dDevice.SetTexture(1, d3dBumpMap)
Call g_d3dDevice.SetTexture(2, d3dEnvMap)
```

After setting the textures to their blending stages, the following code 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 g_d3dDevice.SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE)
Call g_d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call g_d3dDevice.SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE)
Call g_d3dDevice.SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1)
Call g_d3dDevice.SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE)
Call g_d3dDevice.SetTextureStageState(0, D3DTSS_TEXCOORDINDEX, 1)

' Stage 1: the bump map
Call g_d3dDevice.SetTextureStageState(1, D3DTSS_TEXCOORDINDEX, 1)
' Use luminance for this example.
Call g_d3dDevice.SetTextureStageState(1, D3DTSS_COLOROP,
D3DTOP_BUMPENVMAPLUMINANCE)
Call g_d3dDevice.SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call g_d3dDevice.SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT)

' Stage 2: a specular environment map
Call g_d3dDevice.SetTextureStageState(2, D3DTSS_TEXCOORDINDEX, 0)
Call g_d3dDevice.SetTextureStageState(2, D3DTSS_COLOROP, D3DTOP_ADD)
Call g_d3dDevice.SetTextureStageState(2, D3DTSS_COLORARG1, D3DTA_TEXTURE)
Call g_d3dDevice.SetTextureStageState(2, D3DTSS_COLORARG2, D3DTA_CURRENT)
```

Once the blending operations and arguments are set, the following code 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 g_d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVMAT00, 1#)
Call g_d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVMAT01, 0#)
Call g_d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVMAT10, 0#)
Call g_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 g_d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVLSCALE, 0.5)  
Call g_d3dDevice.SetTextureStageStateSingle(1, D3DTSS_BUMPENVLOFFSET, 0#)
```

After your application configures bump mapping parameters, it can render as it normally would, and the rendered polygons will receive bump mapping effects.

Note:

The preceding example shows parameters set for specular environment mapping. When performing diffuse light mapping, applications would set the texture blending operation for the last stage to D3DTOP_MODULATE. For more information, see Diffuse Light Maps.

Billboarding

When creating 3-D scenes, applications 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. Direct3D applications can easily 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 containing the object's image to a rectangular primitive. The primitive is rotated so that it always faces the viewer. 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 their animated sprites. For instance, when the player 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 onto a rectangular primitive. Billboarding is often used in games to render images of trees and bushes.

When an image is applied to a billboard, the rectangular primitive must first be rotated so that the resulting image faces the viewer. Your application must then translate it into position. The program can then apply a texture to the primitive.

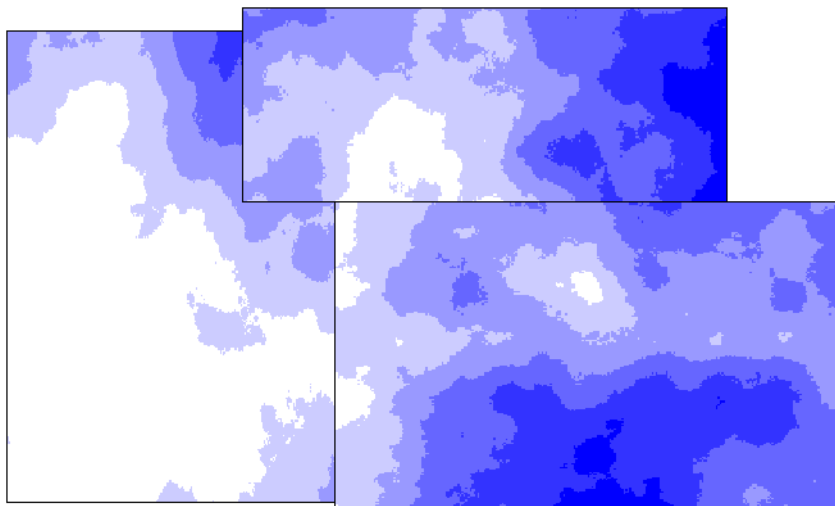
Billboarding works best for symmetrical objects, especially objects that are symmetrical around the vertical axis. It also requires that the altitude of the viewpoint doesn't increase too much. If the user is allowed to view the billboarded from above, it will become readily apparent that the object 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 around two axes instead of one, your application can enable the viewer to look at a billboard from any angle. Typically, your application will rotate the billboard around the horizontal and vertical axes.

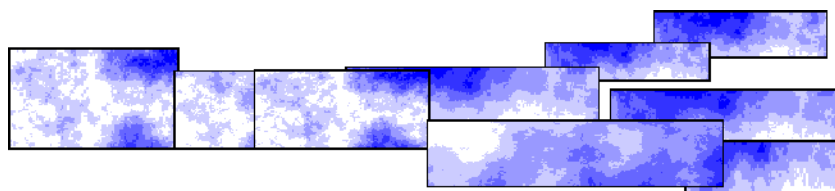
To make a simple cloud, your application can rotate a rectangular primitive around one or two axes so that it faces the viewer. A cloud-like texture can then be applied to the primitive with transparency. For details on applying transparent textures to primitives, see Texture Blending. The cloud can be animated by applying a series of textures over time.

Applications 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 be moved accordingly. More billboards may need to be added as the plume rises and disperses.

A vapor trail is just a smoke plume that doesn't rise. However, like a smoke plume, it does disperse over time. The following figure illustrates the technique of using billboards to simulate a vapor trail.



Fire, Flares, Explosions, and More

Applications can use Direct3D to simulate natural phenomena involving energy releases. For instance, a program can generate the appearance of fire by applying flame-like textures to a set of billboards. This is especially effective if the program 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.

Flares and flashes can be simulated by applying successively brighter light maps to all primitives in a scene. Although this is a computationally high-overhead technique, it allows your program to simulate a localized flare or flash. That is, the portion of the scene where the flare or flash originates can be brightened first.

Another technique is to position a billboard in front of the scene so that the entire render target area is covered. The program applies successively whiter textures to the billboard and decreases the transparency over time. The entire scene will fade 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 procedures like those used for fire, flashes, and flares. For instance, your program 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 will allow the viewer 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 and Current Texture](#).

Motion Blur

The perceived speed of an object in a 3-D scene can be enhanced by blurring the object, and by leaving a blurred trail of object images behind the object. Direct3D applications can accomplish this by rendering the object multiple times per frame.

[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 **DirectDrawSurface7.Flip** method. Your Direct3D program 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 of its textures nearly transparent. Each time the object is rendered, the transparency of the object's texture is decreased. When your program 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.

[C++]

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 **IDirectDrawSurface7::Flip** method to display the frame on the screen.

[Visual Basic]

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 **DirectDrawSurface7.Flip** method to display the frame on the screen.

If your program 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 would render the entire scene multiple times per frame. Each time the scene is rendered, your program must move the viewpoint slightly. If your scene is highly complex, your user may see a visible performance degradation as acceleration is increased because of the increasing number of scene renderings per frame.

Stencil Buffer Techniques

Applications use the stencil buffer to mask pixels in an image. The mask controls whether or not the pixel is drawn. For more information on the stencil buffer, see Stencil Buffers.

Direct3D applications can achieve a wide range of special effects with the stencil buffer. Some of the more common effects are listed here. This list is by no means exhaustive.

- Dissolves, Fades, and Swipes
- Decaling
- Compositing
- Outlines and Silhouettes

Dissolves, Fades, and Swipes

Applications are increasingly employing 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 Direct3D provides methods of using multiple texture blending to achieve the same effect, applications that use the stencil buffer for dissolves can utilize the 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 such that most of the pixels from the starting image pass the test. Most of the pixels of 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 just 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. When an application performs, for instance, a left-to-right swipe. The ending image appears to gradually slide on top of the starting image from left to right. Just as in doing 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 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 left to right, your application must read pixels from the ending image from right to left.

Decaling

Direct3D applications use the technique of decaling to control which pixels from a particular primitive image are drawn to the rendering target surface. Programs apply decals to the images of primitives to enable coplanar polygons to be rendered 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 out 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 together with the driver's forward view.

Outlines and Silhouettes

The stencil buffer can be used for more abstract effects, such as outlining and silhouetting.

If your program applies a stencil mask to the image of a primitive that is the same shape, but slightly smaller than the primitive, the resulting image will only contain the

primitive's outline. It 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 will contain a "hole" where the primitive should be. Your program can then fill the "hole" with black to produce a silhouette of the primitive.

Colored Lights

[C++]

The color-related members of the **D3DLIGHT7** structure (**d3dDiffuse**, **d3dSpecular**, **d3dAmbient**) are **D3DCOLORVALUE** structures. The colors defined by these structures 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 could create a strong light that washes out a scene by setting the color to large values. You could 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 **D3DLIGHT7** 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 could create a strong light that washes out a scene by setting the color to large values. You could 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.

Antialiasing

Antialiasing is a technique you can use to reduce the appearance of "jaggies"—the stair-step pixels used to draw any line that isn't 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.

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

Note

If your application will use antialiasing, it should create the render-target surface with the DDSCAPS2_HINTANTIALIASING capability flag. Applications that omit the flag might suffer from poor performance when antialiasing is enabled, or might not receive antialiasing services at all.

Edge Antialiasing

In edge antialiasing, you render a scene, then re-render the convex silhouettes of the objects to be antialiased with lines. The system redraws those edges, blurring them to reduce artifacts.

[C++]

If a device supports edge antialiasing, it exposes the D3DPRASERCAPS_ANTIALIASEDGES capability flag in the **D3DPRIMCAPS** structure (filled by calling the **IDirect3DDevice7::GetCaps** method). If it does, set the D3DRENDERSTATE_EDGEANTIALIAS render state to TRUE, then redraw only the edges in the scene, using **IDirect3DDevice7::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.

[Visual Basic]

If a device supports edge antialiasing, it exposes the D3DPRASERCAPS_ANTIALIASEDGES capability flag in the **D3DPRIMCAPS** type (filled by calling the **Direct3DDevice7.GetCaps** method). If it does, set the D3DRENDERSTATE_EDGEANTIALIAS render state to True, then redraw only the edges in the scene, using **Direct3DDevice7.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 color (for example, silhouette edges) or boundaries between very different materials. Antialiasing the edge between two polygons of roughly the same color will have no effect, yet is still computationally expensive. For these reasons, full-scene antialiasing is often preferred, given that the current hardware supports it. 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. Note that full-scene antialiasing, when supported, only affects triangles and groups of triangles; lines cannot be antialiased by using Direct3D services.

[C++]

On some hardware, full-scene antialiasing can be applied only when the application renders the polygons sorted from back to front. To find out whether this is true for the current device, retrieve the device capabilities by calling

IDirect3DDevice7::GetCaps method. After the call, check the **dwRasterCaps** member of the associated **D3DPRIMCAPS** structures. If the device requires back to front rendering for antialiasing, it exposes the **D3DPRASTERCAPS_ANTIASSORTDEPENDENT** capability flag in **dwRasterCaps**. If the device can perform antialiasing without regard to polygon order, it will expose the **D3DPRASTERCAPS_ANTIASSORTINDEPENDENT** flag. Of course, the absence of both flags indicates that the device cannot perform full-scene antialiasing at all.

After finding out whether or not you need to sort the polygons, set the **D3DRENDERSTATE_ANTI_ALIAS** render state to **D3DANTIALIAS_SORTDEPENDENT** or **D3DANTIALIAS_SORTINDEPENDENT** and draw the scene.

When you no longer need full-scene antialiasing, disable it by setting **D3DRENDERSTATE_ANTI_ALIAS** to **D3DANTIALIAS_NONE**.

[Visual Basic]

On some hardware, full-scene antialiasing can be applied only when the application renders the polygons sorted from back to front. To find out whether this is true for the current device, retrieve the device capabilities by calling **Direct3DDevice7.GetCaps** method. After the call, check the **IRasterCaps** member of the associated **D3DPRIMCAPS** types. If the device requires back to front rendering for antialiasing, it exposes the **D3DPRASTERCAPS_ANTIASSORTDEPENDENT** capability flag in **IRasterCaps**. If the device can perform antialiasing without regard to polygon order, it will expose the **D3DPRASTERCAPS_ANTIASSORTINDEPENDENT** flag. Of course, the absence of both flags indicates that the device cannot perform full-scene antialiasing at all.

After finding out whether or not you need to sort the polygons, set the **D3DRENDERSTATE_ANTI_ALIAS** render state to **D3DANTIALIAS_SORTDEPENDENT** or **D3DANTIALIAS_SORTINDEPENDENT** and draw the scene.

When you no longer need full-scene antialiasing, disable it by setting **D3DRENDERSTATE_ANTI_ALIAS** to **D3DANTIALIAS_NONE**.

GUIDs

[C++]

Direct3D uses globally unique identifiers, or GUIDs, to identify parts of the interface. When you use the **QueryInterface** method to determine whether an object supports an interface, you identify the interface you're interested in by using its GUID.

To use GUIDs successfully in your application, you must either define INITGUID prior to all other include and define statements, or you must link to the Dxguid.lib library. You should define INITGUID in only one of your source modules.

Note that you use GUIDs differently depending on whether your application is written in C or C++. In C, you pass a pointer to the GUID (&IID_IDirect3D, for example), but in C++, you pass a reference to it (simply IID_IDirect3D).

[Visual Basic]

Like all of DirectX, the Direct3D component uses globally unique identifiers, or GUIDs, to identify parts of the API. DirectX for Visual Basic uses Strings to represent GUIDs where they are required—for example, when your application creates a **DirectDraw7** object by calling the **DirectX7.DirectDrawCreate** method.

For more information, see Using GUIDs.

Performance Optimization

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 Direct3D application:

- Databases and Culling
- Batching Primitives
- Lighting Tips
- Texture Size
- General Performance Tips
- Z-buffer Performance

Databases and Culling

Building a reliable database of the objects in your world is the key to excellent performance in 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, building low-poly models from the start, and add polygons if you feel that you can do so without sacrificing performance later in the development process. Try to keep the total number of polygons in the neighborhood of 2500. Remember, "the fastest polygons are the ones you don't draw."

Batching Primitives

To get the best rendering performance during execution, you should 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 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

Since 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 just 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 or not they 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 `D3DRENDERSTATE_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 already mentioned, textures should be kept as small as possible.

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.
- If you can use smaller textures, 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 both with hardware acceleration and software emulation.

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 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.

[C++]

You can also improve the performance of your application by performing visibility testing before rendering. Direct3D Immediate Mode offers the **IDirect3DDevice7::ComputeSphereVisibility** method for this purpose. The **ComputeSphereVisibility** method provides information about the visibility of a given set of polygons. If the method reports that the geometry isn't visible, you can avoid even rendering the object.

[Visual Basic]

You can also improve the performance of your application by performing visibility testing before rendering. Direct3D Immediate Mode offers the **Direct3DDevice7.ComputeSphereVisibility** method for this purpose. The **ComputeSphereVisibility** method provides information about the visibility of a given set of polygons. If the method reports that the geometry isn't visible, you can avoid even rendering the object.

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 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.

Troubleshooting

This section lists common categories of problems that you may encounter when writing Direct3D programs, and what you should do to prevent them.

- Device Creation
- Using Lit Vertices
- Nothing Visible
- Debugging
- Borland Floating-Point Initialization
- Parameter Validation

Device Creation

If your application fails during device creation, check for the following common errors:

- You must specify DDSCAPS_3DDEVICE when you create the DirectDraw surface.
- If you're using a palettized device, you must attach the palette.
- If you're using a z-buffer, you must attach it to the rendering target.
- Make sure you check the device capabilities, particularly the render depths.
- Check whether you are using system or video memory.
- Ensure that the registry has not been corrupted.

Using Lit Vertices

[C++]

Applications that use lit vertices, analogous to the **D3DLVERTEX** structure, should disable the Direct3D lighting engine by setting the **D3DRENDERSTATE_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 **D3DVERTEX** don't, by their nature, contain a vertex normal, any color information passed-in to Direct3D will be 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 **D3DRENDERSTATE_LIGHTING** to **FALSE**.

[Visual Basic]

Applications that use lit vertices, analogous to the **D3DLVERTEX** type, should disable the Direct3D lighting engine by setting the **D3DRENDERSTATE_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 **D3DVERTEX** don't, by their nature, contain a vertex normal, any color information passed-in to Direct3D will be 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 **D3DRENDERSTATE_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 will 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 Direct3D application can be challenging. In addition to checking all the return values (a particularly important piece of advice in Direct3D programming, which is so dependent on very different hardware implementations), try the following techniques:

- 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 the preceding 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 the Borland company report floating-point exceptions in a manner that is incompatible with Direct3D. To solve this problem, you should 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 Direct3D Immediate Mode runtime performs more parameter validation than the retail version, which sometimes performs no validation at all. This allows 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 runtime. Applications must conform to these limits, or unpredictable (and highly undesirable) results can occur when running on the retail version of Direct3D. For example, the

IDirect3DDevice7::DrawPrimitive method accepts a parameter (*dwVertexCount*) that indicates the number of vertices that the method will render. The method can only accept values between 0 and 65,535 (0x0000 and 0xFFFF). In the debug version of Direct3D, if you pass 65,536 (one more than the limit), the method will fail 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 runtime, 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.

Direct3D Immediate Mode Tutorials

This section contains a series of tutorials, each providing step-by-step instructions for implementing the basics of Direct3D® Immediate Mode in a C/C++ or Visual Basic® application. The tutorials are written parallel to a set of sample files that are provided with this SDK in the \Samples\Multimedia\D3DIM\Tutorials directory, following their code path and providing explanations along the way. Readers are encouraged to follow along in the sample code as they move through these tutorials.

- Direct3D Immediate Mode C/C++ Tutorials
- Direct3D Immediate Mode Visual Basic Tutorials

Direct3D Immediate Mode C/C++ Tutorials

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

The tutorials in this section show how to use Direct3D 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 here:

- Tutorial 1: Rendering a Single Triangle
- Tutorial 2: Adding a Depth Buffer
- Tutorial 3: Using Alternate Primitive Styles

- Tutorial 4: Using Device Enumeration
- Tutorial 5: Using Texture Maps

Note

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 SDK. Changes are made for brevity only, and are limited to comments to avoid changing the behavior of the code.

Tutorial 1: Rendering a Single Triangle

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

To use Direct3D, you first create an application window, then create and initialize DirectDraw® and Direct3D-related objects. You use the COM interfaces that these objects implement to manipulate them and to create the subordinate objects required to render a scene. The Triangle sample project upon which this tutorial is based illustrates these tasks by rendering the simplest possible scene: a single triangle. The Triangle sample uses the following steps to set up Direct3D, render a scene, and eventually shut down:

- Step 1: Create a Window
- Step 2: Initialize System Objects
- Step 3: Initialize the Scene
- Step 4: Monitor System Messages
- Step 5: Render and Display the Scene
- Step 6: Shut Down

In addition to these steps, Triangle performs some standard tasks common to windowed applications. Although these tasks aren't difficult, the Triangle sample (and all other windowed samples) performs the following tasks as the needed:

- Handle Window Movement
 - Handle Window Resizing
-

Step 1: Create a Window

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

The first thing any Windows® application must do when it is executed is create an application window to display a user interface. Keeping with this, when Triangle begins execution at its **WinMain** function, it uses the following code to perform window initialization:

```
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT )
{
    // Register the window class
    WNDCLASS wndClass = { CS_HREDRAW | CS_VREDRAW, WndProc, 0, 0, hInst,
        LoadIcon( hInst, MAKEINTRESOURCE(IDI_MAIN_ICON)),
        LoadCursor(NULL, IDC_ARROW),
        (HBRUSH)GetStockObject(WHITE_BRUSH), NULL,
        TEXT("Render Window") };
    RegisterClass( &wndClass );

    // Create our main window
    HWND hWnd = CreateWindow( TEXT("Render Window"),
        TEXT("D3D Tutorial: Drawing One Triangle"),
        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
        CW_USEDEFAULT, 300, 300, 0L, 0L, hInst, 0L );
    ShowWindow( hWnd, SW_SHOWNORMAL );
    UpdateWindow( hWnd );
}
```

The preceding code is standard Windows programming, covered here mostly for thoroughness. The sample starts by defining and registering a window class called "Render Window." The window class is defined to redraw on size events, to use an application-provided icon as a resource, and to have a white background. After the class is registered, the code creates a basic top-level window that uses the registered class, with a client area of 300 pixels wide by 300 pixels tall, and has no menu or child windows. The sample uses the WS_OVERLAPPEDWINDOW window style to create a window that includes minimize, maximize, and close boxes common to windowed applications. (If the sample were to run in full-screen mode, the preferred window style is WS_EX_TOPMOST.) Once the window is created, the code calls standard Win32® functions to show and update the window.

With the application window ready, you can begin setting up the essential DirectX® objects, which is the topic of Step 2: Initialize System Objects.

Step 2: Initialize System Objects

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

After you create an application window, you can begin initializing the primary DirectX objects whose services you will draw on to render the scene. For a Direct3D application, this means creating and configuring DirectDraw, rendering surfaces, a rendering device, and a viewport. For clarity, the Triangle sample separates system object initialization code from the code for initializing the scene. As a result, geometry, application-specific data structures, and lesser Direct3D objects like materials are initialized with the scene. This isn't a requirement, but it makes for simpler code.

The Triangle sample performs system initialization in the Initialize3DEnvironment application-defined function, called from **WinMain** after the window is created. Although preparing these objects isn't complex, the code is a little too lengthy to discuss in one place. As a result, the steps taken by the Initialize3DEnvironment function are presented in the following substeps:

- Step 2.1: Initialize DirectDraw
- Step 2.2: Set Up DirectDraw Surfaces
- Step 2.3: Initialize Direct3D
- Step 2.4: Prepare the Viewport

Note

The Triangle sample code performs initialization by calling methods from within the **WinMain** function, immediately after the application window is created, rather than in response to system creation messages such as WM_CREATE. It does this to avoid relying on system message ordering, which can differ across platforms.

Step 2.1: Initialize DirectDraw

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

After creating the application window, the first object you will create is the **DirectDraw** object, which is required to set your application's cooperative level, and to create the surfaces for display and for use as the render target of a rendering device.

The Triangle sample starts performing initialization by creating a **DirectDraw** object and setting the application's cooperative level, as shown in the following code:

```
HRESULT Initialize3DEnvironment( HWND hWnd )
{
    HRESULT hr;

    hr = DirectDrawCreateEx( NULL, (VOID**)&g_pDD, IID_IDirectDraw7, NULL );
    if( FAILED( hr ) )
        return hr;
```

The preceding code creates a **DirectDraw** object by calling the **DirectDrawCreateEx** **DirectDraw** global function. It passes **NULL** in the first parameter to request that the function create a **DirectDraw** object for the active display driver. For hardware that doesn't support GDI, such as 3-D only hardware, you should explicitly specify the globally unique identifier (GUID) of the desired driver in the first parameter. These GUIDs are normally obtained through enumeration. The second parameter is the address of a global variable that **DirectDrawCreateEx** fills with the address of the **IDirectDraw7** interface for the **DirectDraw** object, and the last parameter is set to **NULL** to indicate that the new object will not be used with COM aggregation features.

The sample continues by setting the application's cooperative level, as follows:

```
hr = g_pDD->SetCooperativeLevel( hWnd, DDSCL_NORMAL );
if( FAILED( hr ) )
    return hr;
```

The sample sets the cooperative level by calling the **IDirectDraw7::SetCooperativeLevel** method. Setting the cooperative level effectively tells the system whether or not the application will render in full-screen mode or in a window. (Note that some hardware cannot render into a window. You can detect such hardware by checking for the absence of the **DDCAPS2_CANRENDERWINDOWED** capability flag when you call **IDirectDraw7::GetCaps**.) The code requests windowed cooperative level, also called the "normal" cooperative level, by including the **DDSCL_NORMAL** in the second parameter it passes to **SetCooperativeLevel**. The **SetCooperativeLevel** method can fail if another application already controls owns full-screen, exclusive mode.

Note

You can include the **DDSCL_FPUSETUP** cooperative level flag to increase performance. For more information about this cooperative level flag, see **DirectDraw Cooperative Levels and FPU Precision**. For general information, see **Cooperative Levels in the DirectDraw documentation**.

Once you create the DirectDraw object and set the cooperative level, you're ready to prepare the surfaces that will be used to contain and display a rendered scene. The Triangle sample does this as discussed in Step 2.2: Set Up DirectDraw Surfaces.

Step 2.2: Set Up DirectDraw Surfaces

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

After you create a DirectDraw object and set the cooperative level, you can create the surfaces that your application will use to render and display a scene. Exactly how you create your surfaces depends largely on whether or not your application will run in a window or in full-screen mode.

Full-screen Application Note

Applications that will run in full-screen mode can create surfaces as shown in the preceding code examples. More often, these applications should take advantage of page-flipping, a feature only available in full-screen, exclusive mode. In this case, instead of explicitly creating two surfaces, you can create a flipping chain of surfaces with a single call. For more information, see *Creating Complex Surfaces and Flipping Chains*.

The Triangle sample, designed to run in a window, starts by creating a primary surface, which represents the display:

```
DDSURFACEDESC2 ddsd;
ZeroMemory( &ddsd, sizeof(DDSURFACEDESC2) );
ddsd.dwSize      = sizeof(DDSURFACEDESC2);
ddsd.dwFlags     = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

// Create the primary surface.
hr = g_pDD->CreateSurface( &ddsd, &g_pddsPrimary, NULL );
if( FAILED( hr ) )
    return hr;
```

The description for the primary surface doesn't contain information about dimensions or pixel format, as these traits are assumed to be the same as the display mode. If the current display mode is 800x600, 16-bit color, DirectDraw ensures that the primary surface matches. After creating the primary surface, you can create the render target surface. In the case of Triangle, this is a separate off-screen surface created as follows:

```
ddsd.dwFlags     = DDSD_WIDTH | DDSD_HEIGHT | DDSD_CAPS;
```

```
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_3DDEVICE;
```

```
// Set the dimensions of the back buffer. Note that if our window changes
// size, we need to destroy this surface and create a new one.
GetClientRect( hWnd, &g_rcScreenRect );
GetClientRect( hWnd, &g_rcViewportRect );
ClientToScreen( hWnd, (POINT*)&g_rcScreenRect.left );
ClientToScreen( hWnd, (POINT*)&g_rcScreenRect.right );
ddsd.dwWidth = g_rcScreenRect.right - g_rcScreenRect.left;
ddsd.dwHeight = g_rcScreenRect.bottom - g_rcScreenRect.top;
```

```
// Create the backbuffer. The most likely reason for failure is running
// out of video memory. (A more sophisticated app should handle this.)
hr = g_pDD->CreateSurface( &ddsd, &g_pddsBackBuffer, NULL );
if( FAILED( hr ) )
    return hr;
```

The preceding code creates an off-screen surface that is equal to the dimensions of the program window. There is no need to create a larger surface, because the dimensions of the window dictate what is visible to the user. (This code also initializes two global variables that are later used to set up the viewport and track the application window size and position.) As the preceding excerpt shows, you must include the DDSCAPS_3DDEVICE capability for any surface that will be used as a render target. This capability causes the system to allocate additional internal data structures that are used only for 3-D rendering. As when creating the primary surface, the pixel format for the off-screen surface is assumed to be the same as the display mode when it isn't provided in the surface description.

Note

Applications that will use a depth buffer should create one and attach it to the render target surface at this point. For simplicity, this tutorial doesn't employ a depth buffer, but they are covered in Tutorial 2: Adding a Depth Buffer and in Depth Buffers.

After creating the primary and render target surface, you can create and attach a DirectDrawClipper object to the display surface. Using a clipper frees you from attempting to handle cases when the window is partially obscured by other windows, or when the window is partially outside the display area. Clippers aren't needed for applications that run in full-screen mode. The Triangle sample uses the following code to create a clipper and associate it with the display window:

```
LPDIRECTDRAWCLIPPER pcClipper;
hr = g_pDD->CreateClipper( 0, &pcClipper, NULL );
if( FAILED( hr ) )
    return hr;

// Assign it to the window handle, then set
// the clipper to the desired surface.
```

```
pcClipper->SetHWnd( 0, hWnd );
g_pddsPrimary->SetClipper( pcClipper );
pcClipper->Release();
```

Having created the basic DirectDraw objects, you can move on to setting up the essential Direct3D objects that will render the scene. The Triangle sample performs this task in Step 2.3: Initialize Direct3D.

Step 2.3: Initialize Direct3D

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

After you create the surfaces your application will need to render and display a scene, you can begin initializing Direct3D objects by retrieving a pointer to the **IDirect3D7** interface for the DirectDraw object, which is used to create all the objects you'll need to render a scene. Note that this interface is exposed by the DirectDraw object, and represents a separate set of features, not a separate object. You retrieve the **IDirect3D7** interface by calling the **IUnknown::QueryInterface** method of the DirectDraw object. The following code from Triangle performs this task:

```
// Query DirectDraw for access to Direct3D
g_pDD->QueryInterface( IID_IDirect3D7, (VOID**)&g_pD3D );
if( FAILED( hr ) )
    return hr;
```

After retrieving a pointer to the **IDirect3D7** interface, you can create a rendering device by calling the **IDirect3D7::CreateDevice** method. The **CreateDevice** method accepts the globally unique identifier (GUID) of the desired device, the address of the **IDirectDrawSurface7** interface for the surface that the device will render to, and the address of a variable that the method will set to an **IDirect3DDevice7** interface pointer if the device object is created successfully. Although the tutorial uses hard-coded GUID values, a real application should enumerate devices to get a GUID. For information about device enumeration, see Enumerating Direct3D Devices.

(The Triangle sample checks the display mode prior to creating the device. If the display is set to a palettized mode, it exits. Attempting to create a device for a palettized surface that doesn't have an associated palette will cause the **CreateDevice** method to fail. This is done for simplicity. A real-world application should create a render target surface and attach a palette, or require that the user set their display mode to 16-bit color or higher.)

The following code, taken from Triangle, checks the display mode, and creates a rendering device:

```
// Check the display mode, and
```

```

ddsd.dwSize = sizeof(DDSURFACEDESC2);
g_pDD->GetDisplayMode( &ddsd );
if( ddsd.ddpfPixelFormat.dwRGBBitCount <= 8 )
    return DDERR_INVALIDMODE;

// The GUID here is hard coded. In a real-world application
// this should be retrieved by enumerating devices.
hr = g_pD3D->CreateDevice( IID_IDirect3DHALDevice, g_pddsBackBuffer,
                        &g_pd3dDevice );
if( FAILED( hr ) )
{
    // If the hardware GUID doesn't work, try a software device.
    hr = g_pD3D->CreateDevice( IID_IDirect3DRGBDDevice, g_pddsBackBuffer,
                            &g_pd3dDevice );
    if( FAILED( hr ) )
        return hr;
}

```

The **CreateDevice** method can fail for many reasons. The most likely cause is when the primary display device doesn't support 3-D features. Another possibility is if the display hardware cannot render in the current display mode. These possibilities should be checked during device enumeration. To keep the code simple, Triangle attempts to create a software rendering device if the hardware device cannot be created.

Note

Even though the **CreateDevice** method accepts a pointer to a `DirectDrawSurface` object, a rendering device is not a surface. Rather, it is a discrete COM object that uses a surface to contain graphics for a rendered scene.

After the device is created, you can create a viewport and assign it to the device, as described in Step 2.4: Prepare the Viewport.

Step 2.4: Prepare the Viewport

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

After you create a rendering device, you can create a viewport and assign it to the device. In short, the viewport determines how the geometry in a 3-D scene is clipped and then represented in the 2-D space of a display screen. (For a conceptual overview about viewports, see Viewports and Clipping.)

Setting up a viewport is a straight-forward process that starts with preparing the viewport parameters in a **D3DVIEWPORT7** structure. The Triangle sample sets the viewport parameters to the dimensions of the render target surface:

```
// Create the viewport
DWORD dwRenderWidth = g_rcScreenRect.right - g_rcScreenRect.left;
DWORD dwRenderHeight = g_rcScreenRect.bottom - g_rcScreenRect.top;
D3DVIEWPORT7 vp = { 0, 0, dwRenderWidth, dwRenderHeight, 0.0f, 1.0f };
```

Once the viewport structure is prepared, the Triangle application assigns it to the rendering device:

```
hr = g_pd3dDevice->SetViewport( &vp );
if( FAILED( hr ) )
    return hr;
```

Now that the basic DirectX objects have been created, you can start preparing the subordinate objects required to render scene, which is the topic of Step 3: Initialize the Scene.

Step 3: Initialize the Scene

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

After creating the primary Direct3D-related objects—a DirectDraw object, a rendering device, and a viewport—you can begin initializing the scene by setting up geometry, preparing materials, and configuring the transformation matrices your application will use. The Triangle sample uses an application-defined function, `App_InitDeviceObjects`, to perform the following steps that initialize the scene:

- Step 3.1: Prepare Geometry
 - Step 3.2: Set Up Material and Initial Lighting States
 - Step 3.3: Prepare and Set Transformation Matrices
-

Step 3.1: Prepare Geometry

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

After you create the primary system objects used with DirectDraw and Direct3D, you can begin initializing the scene. The Triangle sample takes this opportunity to initialize geometry by defining vertices in an array of **D3DVERTEX** structures. Technically, you aren't required to set up the geometry at this time—you can do it anytime prior to calling rendering methods:

```
HRESULT App_InitDeviceObjects( LPDIRECT3DDEVICE7 pd3dDevice )
{
    // Data for the geometry of the triangle. Note that this tutorial only
    // uses ambient lighting, so the vertices' normals are not actually used.
    D3DVECTOR p1( 0.0f, 3.0f, 0.0f );
    D3DVECTOR p2( 3.0f,-3.0f, 0.0f );
    D3DVECTOR p3(-3.0f,-3.0f, 0.0f );
    D3DVECTOR vNormal( 0.0f, 0.0f, 1.0f );

    // Initialize the 3 vertices for the front of the triangle
    g_pvTriangleVertices[0] = D3DVERTEX( p1, vNormal, 0, 0 );
    g_pvTriangleVertices[1] = D3DVERTEX( p2, vNormal, 0, 0 );
    g_pvTriangleVertices[2] = D3DVERTEX( p3, vNormal, 0, 0 );

    // Initialize the 3 vertices for the back of the triangle
    g_pvTriangleVertices[3] = D3DVERTEX( p1, -vNormal, 0, 0 );
    g_pvTriangleVertices[4] = D3DVERTEX( p3, -vNormal, 0, 0 );
    g_pvTriangleVertices[5] = D3DVERTEX( p2, -vNormal, 0, 0 );
}
```

The preceding code fragment defines three points in 3-D space that define a triangle that sits upright in the z=0 plane. After defining the geometry to be displayed, the Triangle sample prepares material and lighting parameters in Step 3.2: Set Up Material and Initial Lighting States.

Step 3.2: Set Up Material and Initial Lighting States

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

After you create the basic 3-D rendering objects (a DirectDraw object, a rendering device, and a viewport), you've got almost all you need to render a simple scene. The next thing to do is to create and configure a material and set some initial lighting states. These can all be changed later if needed. For an introduction to these concepts, see Lighting and Materials.

The Triangle sample sets material parameters, as shown here:

```
D3DMATERIAL7 mtrl;  
ZeroMemory( &mtrl, sizeof(mtrl) );  
mtrl.ambient.r = 1.0f;  
mtrl.ambient.g = 1.0f;  
mtrl.ambient.b = 0.0f;  
pd3dDevice->SetMaterial( &mtrl );
```

The preceding code prepares material properties, represented by a **D3DMATERIAL7** structure. The code prepares the **D3DMATERIAL7** structure to describe a material that will reflect the red and green components of ambient light, making it appear yellow in the scene. (This tutorial only uses ambient light, so it only sets an ambient reflectance property. A real-world application would use direct light as well as ambient light, and should therefore set diffuse and specular reflectance properties as well.) After preparing the material properties, the code applies them in the device by calling the **IDirect3DDevice7::SetMaterial** method.

Note

When using textures, the object material is usually omitted or colored white.

After the current material is selected, all polygons will be rendered using this material. However, before anything in the scene will be visible you need to provide some light. The code in the Triangle application sets an ambient light level by calling the **IDirect3DDevice7::SetRenderState** method with the **D3DRENDERSTATE_AMBIENT** render state for white ambient light:

```
// The ambient lighting value is another state to set. Here, we are turning  
// ambient lighting on to full white.  
pd3dDevice->SetRenderState( D3DRENDERSTATE_AMBIENT, 0xffffffff );
```

Now that the geometry, material, and initial lighting parameters are set, the sample moves on to setting up the transformation matrices. This is covered in Step 3.3: Prepare and Set Transformation Matrices.

Step 3.3: Prepare and Set Transformation Matrices

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Another step in setting up a simple scene involved setting the world, view, and projection matrices. The system applies these matrices to geometry to place it in the scene, adjust for the camera's location and orientation, and scale vertex data to make distant objects appear smaller than near objects. (For a conceptual overview, see the Geometry Pipeline.)

The Triangle sample starts by creating an identity matrix in a **D3DMATRIX** structure, then it manipulates the matrix to produce the desired transformations. Once a matrix is ready, the code assigns it to the device by calling the **IDirect3DDevice7::SetTransform** method with the **D3DTRANSFORMSTATE_WORLD**, **D3DTRANSFORMSTATE_VIEW**, or **D3DTRANSFORMSTATE_PROJECTION** values:

```
// Start by setting up an identity matrix.
D3DMATRIX mat;
mat._11 = mat._22 = mat._33 = mat._44 = 1.0f;
mat._12 = mat._13 = mat._14 = mat._41 = 0.0f;
mat._21 = mat._23 = mat._24 = mat._42 = 0.0f;
mat._31 = mat._32 = mat._34 = mat._43 = 0.0f;

// The world matrix controls the position and orientation
// of the polygons in world space. We'll use it later to
// spin the triangle.
D3DMATRIX matWorld = mat;
pd3dDevice->SetTransform( D3DTRANSFORMSTATE_WORLD, &matWorld );

// The view matrix defines the position and orientation of
// the camera. Here, we are just moving it back along the z-
// axis by 10 units.
D3DMATRIX matView = mat;
matView._43 = 10.0f;
pd3dDevice->SetTransform( D3DTRANSFORMSTATE_VIEW, &matView );

// The projection matrix defines how the 3-D scene is "projected"
// onto the 2-D render target surface. For more information,
// see "What Is the Projection Transformation?"

// Set up a very simple projection that scales x and y
// by 2, and translates z by -1.0.
D3DMATRIX matProj = mat;
matProj._11 = 2.0f;
matProj._22 = 2.0f;
matProj._34 = 1.0f;
matProj._43 = -1.0f;
matProj._44 = 0.0f;
pd3dDevice->SetTransform( D3DTRANSFORMSTATE_PROJECTION, &matProj );
```

After setting the transformations, Triangle is done setting up the scene. The **App_InitDeviceObjects** application-defined function returns **S_OK** to the caller, the **Initialize3DEnvironment** application-defined function. The **Initialize3DEnvironment** function then returns that value to **WinMain**, which moves on to process system messages, as discussed in Step 4: Monitor System Messages.

Step 4: Monitor System Messages

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

After you've created the application window, created the DirectX objects, then initialized the scene, you're ready to render the scene. In most cases Windows applications monitor system messages in their message loop, and render frames whenever no messages are in queue. The Triangle sample is no different; it uses the following code for its message loop:

```

BOOL bGotMsg;
MSG msg;
PeekMessage( &msg, NULL, 0U, 0U, PM_NOREMOVE );
g_bReady = TRUE;

while( WM_QUIT != msg.message )
{
    // Use PeekMessage() if the app is active, so we can use idle time to
    // render the scene. Else, use GetMessage() to avoid eating CPU time.
    if( g_bActive )
        bGotMsg = PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE );
    else
        bGotMsg = GetMessage( &msg, NULL, 0U, 0U );

    if( bGotMsg )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    else
    {
        // Render a frame during idle time (no messages are waiting)
        if( g_bActive && g_bReady )
            Render3DEnvironment();
    }
}

```

This code uses a global flag variable, *g_bActive*, to keep track of when it's active, and another variable, *g_bReady*, to indicate that all system objects are ready to render a scene. The application sets *g_bActive* to FALSE whenever the window isn't visible, and it sets the *g_bReady* variable to FALSE whenever it needs to recreate the objects used to render the scene. (The latter situation is covered in Handle Window Resizing.)

If the application is active, it checks the message queue to see if there are any pending messages. If there are messages in queue, the code dispatches them like any other Windows application. Otherwise, it calls the `Render3DEnvironment` application-defined function to render a frame of the scene, which is the topic of Step 5: Render and Display the Scene.

Step 5: Render and Display the Scene

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Whenever your application isn't processing system messages, it can render a frame of a scene. The Triangle sample renders the scene in the `Render3DEnvironment` application-defined function called from **WinMain** whenever the message queue is empty. The `Render3DEnvironment` function subdivides the task into three substeps:

- Step 5.1: Update the Scene
 - Step 5.2: Render the Scene
 - Step 5.3: Update the Display
-

Step 5.1: Update the Scene

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Immediately after it is called, the `Render3DEnvironment` application-defined function in the Triangle sample calls `App_FrameMove` (another application-defined function). The `App_FrameMove` function simply updates the world matrix that Direct3D applies to the geometry to reflect a rotation around the y-axis based on an internal count value, passed to the function in the *fTimeKey* parameter. Because the rotation is applied once per frame, the end result looks like the model is rotating in place.

```
HRESULT App_FrameMove( LPDIRECT3DDEVICE7 pd3dDevice, FLOAT fTimeKey )
{
    // For this simple tutorial, we are rotating the triangle about the y-axis.
    // To do this, just set up a 4x4 matrix defining the rotation, and set it
    // as the new world transform.
```

```

D3DMATRIX matSpin;
matSpin._11 = matSpin._22 = matSpin._33 = matSpin._44 = 1.0f;
matSpin._12 = matSpin._13 = matSpin._14 = matSpin._41 = 0.0f;
matSpin._21 = matSpin._23 = matSpin._24 = matSpin._42 = 0.0f;
matSpin._31 = matSpin._32 = matSpin._34 = matSpin._43 = 0.0f;

matSpin._11 = (FLOAT)cos( fTimeKey );
matSpin._33 = (FLOAT)cos( fTimeKey );
matSpin._13 = -(FLOAT)sin( fTimeKey );
matSpin._31 = (FLOAT)sin( fTimeKey );

pd3dDevice->SetTransform( D3DTRANSFORMSTATE_WORLD, &matSpin );

return S_OK;
}

```

In the real world, of course, your applications will do much more than apply a single rotation on a single model. (For more information rotation matrices, see Rotation in the 3-D Transformations section.)

After you update the geometry in the scene, you can render it to the render target surface, as the Triangle sample does in Step 5.2: Render the Scene.

Step 5.2: Render the Scene

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Once the geometry has been updated to reflect the desired animation, you can render the scene. The Triangle sample takes a typical approach: the App_Render application-defined function called from the sample's Render3DEnvironment function starts by clearing the viewport:

```

HRESULT App_Render( LPDIRECT3DDEVICE7 pd3dDevice, D3DRECT* prcViewRect )
{
    // Clear the viewport to a blue color.
    pd3dDevice->Clear( 1UL, prcViewRect, D3DCLEAR_TARGET, 0x000000ff,
        0L, 0L );
}

```

The preceding code calls the **IDirect3DDevice7::Clear** method to clear the viewport. The first two methods that the **Clear** method accepts are the address of an array of rectangles that describe the areas on the render target surface to be cleared, and a value that informs the method how many rectangles from the array should be cleared. In most cases, you'll use a single rectangle that covers the entire render target. The third parameter determines the method's behavior. It can clear a render-target surface,

an associated depth buffer, the stencil buffer, or any combination of the three. Because this tutorial doesn't use a depth buffer, 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 Triangle sample sets the clear color for the render target surface to blue. Because the remaining parameters aren't used in this tutorial, the code just sets them to zero. The **Clear** method will ignore them when the corresponding flag isn't present.

After clearing the viewport, the Triangle sample informs Direct3D that rendering will begin, renders the scene, then signals that rendering is complete, as shown here:

```
// Begin the scene
if( FAILED( pd3dDevice->BeginScene() ) )
    return E_FAIL;

// Draw the triangle using a DrawPrimitive() call. Subsequent
// tutorials will go into more detail on the various calls for
// drawing polygons.
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, D3DFVF_VERTEX,
                          g_pvTriangleVertices, 6, NULL );

// End the scene.
pd3dDevice->EndScene();

return S_OK;
}
```

The **IDirect3DDevice7::BeginScene** and **IDirect3DDevice7::EndScene** methods signal to the system when rendering is beginning or is complete. You can only call rendering methods between calls to these methods. Even if rendering methods fail, you should call **EndScene** before calling **BeginScene** again.

After rendering the scene to the off-screen render target, you can update the user's display. The tutorial sample performs this in Step 5.3: Update the Display.

Step 5.3: Update the Display

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Once a scene has been rendered to the render-target surface, you can show the results on screen. A windowed application usually does this by blitting the content of the render-target surface to the primary surface, and a full-screen application that employs page-flipping would simply flip the surfaces in the flipping chain. The

Triangle sample uses the former method because it runs in a window, using the following code:

```
HRESULT ShowFrame()
{
    if( NULL == g_pddsPrimary )
        return E_FAIL;

    // We are in windowed mode, so perform a blit from the backbuffer to the
    // correct position on the primary surface
    return g_pddsPrimary->Blit( &g_rcScreenRect, g_pddsBackBuffer,
                               &g_rcViewportRect, DDBLT_WAIT, NULL );
}
```

Note that the preceding application-defined function simply blits the entire contents of the render target surface to the window on the desktop. The tutorial tracks the destination rectangle for the blit in the *g_rcScreenRect* global variable. This rectangle is updated whenever the user moves the window, as covered in the Handle Window Movement section.

Step 6: Shut Down

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

At some point during execution, your application must shut down. Shutting down a DirectX application not only means that you should destroy the application window, but you also deallocate any DirectX objects your application uses and invalidate the pointers to them. The Triangle calls an application-defined function to handle this cleanup, called *Cleanup3DEnvironment*, when it receives a *WM_DESTROY* message:

```
HRESULT Cleanup3DEnvironment()
{
    // Cleanup any objects created for the scene
    App_DeleteDeviceObjects( g_pd3dDevice );

    // Release the DDraw and D3D objects used by the app
    if( g_pD3D )        g_pD3D->Release();
    if( g_pddsBackBuffer ) g_pddsBackBuffer->Release();
    if( g_pddsPrimary )  g_pddsPrimary->Release();

    // Do a safe check for releasing the D3DDEVICE. RefCount should be zero.
```

```

if( g_pd3dDevice )
    if( 0 < g_pd3dDevice->Release() )
        return E_FAIL;

// Do a safe check for releasing DDRAW. RefCount should be zero.
if( g_pDD )
    if( 0 < g_pDD->Release() )
        return E_FAIL;

g_pd3dDevice = NULL;
g_pD3D       = NULL;
g_pddsBackBuffer = NULL;
g_pddsPrimary = NULL;
g_pDD        = NULL;

return S_OK;
}

```

The preceding function deallocates the DirectX objects it uses by calling the **IUnknown::Release** methods for each object. Because the tutorial follows COM rules, the reference counts for most objects should become zero and are automatically removed from memory.

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 DirectX 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.

Handle Window Movement

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

Any DirectX application that runs in a window must track the position of the client area for the window so that blits to the window will appear on the desktop in the right place. Failing to track this results in graphics appearing outside the application window—a confusing sight for the user. The Triangle sample responds to the WM_MOVE messages that the system sends to the window procedure as shown here:

```

.
.
.
case WM_MOVE:

```

```

// Move messages need to be tracked to update the screen rects
// used for blitting the backbuffer to the primary.
if( g_bActive && g_bReady )
    OnMove( (SHORT)LOWORD(IParam), (SHORT)HIWORD(IParam) );
break;

```

The *g_bActive* variable is set elsewhere in the window procedure to reflect whether or not the window is active—when the window is minimized it's set to FALSE. The *g_bReady* variable is TRUE except when the application is in the midst of re-creating the DirectX objects it uses. If these variables are both TRUE, the OnMove application-defined function gets called:

```

VOID OnMove( INT x, INT y )
{
    DWORD dwWidth = g_rcScreenRect.right - g_rcScreenRect.left;
    DWORD dwHeight = g_rcScreenRect.bottom - g_rcScreenRect.top;
    SetRect( &g_rcScreenRect, x, y, x + dwWidth, y + dwHeight );
}

```

This function simply recalculates the size of the global variable, *g_rcScreenRect*, that the tutorial uses as the destination rectangle when it updates the display in Step 5.3: Update the Display.

Handle Window Resizing

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

Any DirectX application that runs in a window must respond to any WM_SIZE messages that the system sends. The render target surface is usually kept as small as possible to conserve memory, and the smallest size is usually the size of the window client area. When window size increases, you must destroy the render target surface and the associated objects and re-create it at an appropriate size. Technically, an application could do this only when the window gets larger, and respond to situations when window size decreases by adjusting the viewport and decreasing the size of the blit accordingly. For simplicity, this tutorial destroys and re-creates the objects it uses whenever it receives a WM_SIZE message:

```
case WM_SIZE:
    // Check to see if we are losing or gaining our window.
    // Set the active flag to match.
    if( SIZE_MAXHIDE==wParam || SIZE_MINIMIZED==wParam )
        g_bActive = FALSE;
    else g_bActive = TRUE;

    // A new window size will require a new back buffer size. The
    // easiest way to achieve this is to release and re-create
    // everything. Note: if the window gets too big, we may run out
    // of video memory and need to exit. This simple app exits
    // without displaying error messages, but a real app would
    // behave itself much better.
    if( g_bActive && g_bReady )
    {
        g_bReady = FALSE;
        if( FAILED( Cleanup3DEnvironment() ) )
            DestroyWindow( hWnd );
        if( FAILED( Initialize3DEnvironment( hWnd ) ) )
            DestroyWindow( hWnd );
        g_bReady = TRUE;
    }
    break;
.
.
.
```

Note that the preceding code sets a global variable to communicate to other portions of the code that the DirectX objects in use are being invalidated. In addition, this code calls the application-defined `Cleanup3DEnvironment` function to destroy the objects, which is also called during application shut-down. Ending the application is discussed in Step 6: Shut Down.

Tutorial 2: Adding a Depth Buffer

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

Direct3D applications often rely on depth buffers to properly display objects in a scene. For a conceptual overview, see [Depth Buffers](#). To use depth buffering, you

must enumerate supported depth-buffer formats, create a depth-buffer surface, attach it to a render-target surface, and enable depth buffering for the rendering device.

This tutorial parallels the code in the Triangle sample project which uses the most commonly supported type of depth buffer, a z-buffer. The ZBuffer sample performs the following steps to use a z-buffer:

- Step 1: Enumerate Depth-Buffer Formats
- Step 2: Create the Depth Buffer
- Step 3: Attach the Depth Buffer
- Step 4: Enable Depth Buffering

Note

The code in the ZBuffer sample is nearly identical to the code in Triangle. This tutorial focuses only on the depth-buffer code unique to ZBuffer, and does not cover setting up Direct3D, rendering, shutting down, or handling Windows messages. For information on these tasks, see Tutorial 1: Rendering a Single Triangle.

Because some rendering devices require depth buffers to be located in particular places in memory, the system requires that you create and attach the depth-buffer surface to the render-target surface before you create a rendering device.

Step 1: Enumerate Depth-Buffer Formats

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Before you can create a depth buffer you must determine what depth-buffer formats, if any, are supported by the rendering device. Call the

IDirect3D7::EnumZBufferFormats method to enumerate the depth-buffer formats that the device supports. The ZBuffer sample uses the following code to enumerate depth-buffer formats:

```
//-----  
// Create the z-buffer AFTER creating the backbuffer and BEFORE creating  
// the d3ddevice.  
//  
// Note: before creating the z-buffer, apps may want to check the device  
// caps for the D3DPRASTERCAPS_ZBUFFERLESSHSR flag. This flag is true for  
// certain hardware that can do HSR (hidden-surface-removal) without a  
// z-buffer. For those devices, there is no need to create a z-buffer.
```

```
//-----
```

```
DDPIXELFORMAT ddpfZBuffer;
g_pD3D->EnumZBufferFormats( *pDeviceGUID,
    EnumZBufferCallback, (VOID*)&ddpfZBuffer );
```

The **EnumZBufferFormats** method accepts the globally unique identifier (GUID) of the device for which the formats will be enumerated, the address of a callback function, and the address of an arbitrary data structure that will be passed to the callback function. The callback function you provide must conform to the **D3DEnumPixelFormatsCallback** function prototype. The system calls the specified callback function once for each supported depth-buffer format, unless the callback function returns **D3DENUMRET_CANCEL**. The ZBuffer sample processes callbacks as follows:

```
static HRESULT WINAPI EnumZBufferCallback( DDPIXELFORMAT* pddpf,
    VOID* pddpfDesired )
{
    // For this tutorial, we are only interested in z-buffers, so ignore any
    // other formats (e.g. DDPF_STENCILBUFFER) that get enumerated. An app
    // could also check the depth of the z-buffer (16-bit, etc,) and make a
    // choice based on that, as well. For this tutorial, we'll take the first
    // one we get.
    if( pddpf->dwFlags == DDPF_ZBUFFER )
    {
        memcpy( pddpfDesired, pddpf, sizeof(DDPIXELFORMAT) );

        // Return with D3DENUMRET_CANCEL to end the search.
        return D3DENUMRET_CANCEL;
    }

    // Return with D3DENUMRET_OK to continue the search.
    return D3DENUMRET_OK;
}
```

When the system calls the callback function, it passes the address of a **DDPIXELFORMAT** structure that describes the pixel format of the depth buffer. The **dwFlags** member will contain **DDPF_ZBUFFER** for any pixel formats that include depth-buffer bits. If so, the **dwZBufferBitDepth** member includes an integer that represents the number of bits in the pixel format reserved for depth information, and the **dwZBitMask** member masks the relevant bits.

For simplicity, this tutorial only uses z-buffers, which are the most common type of depth buffer. It ignores any other formats (such as **DDPF_STENCILBUFFER**) that the system enumerates. Applications could also check the bit depth of the z-buffer (8-, 16-, 24-, 32-bit) and make a choice based on that as well. If a suitable format is found, the function copies the provided **DDPIXELFORMAT** structure to the address passed in the second parameter (also a **DDPIXELFORMAT** structure), and returns **D3DENUMRET_CANCEL** to stop the enumeration.

After you determine the format of the depth buffer, you can create a `DirectDrawSurface` that uses that format, which is the topic of Step 2: Create the Depth Buffer.

Step 2: Create the Depth Buffer

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Now that you have chosen the depth-buffer format, you can create the `DirectDrawSurface` object that will become the depth buffer. The pixel format of the surface is the one determined through enumeration, but the surface dimensions must be identical to the render-target surface to which it will be attached. The ZBuffer sample uses the following code for this task:

```
// If we found a good zbuffer format, then the dwSize field will be
// properly set during enumeration. Else, we have a problem and will exit.
if( sizeof(DDPIXELFORMAT) != ddpfZBuffer.dwSize )
    return E_FAIL;

// Get z-buffer dimensions from the render target
// Setup the surface desc for the z-buffer.
ddsd.dwFlags      = DDSD_CAPS|DDSD_WIDTH|DDSD_HEIGHT|DDSD_PIXELFORMAT;
ddsd.ddsCaps.dwCaps = DDSCAPS_ZBUFFER;
ddsd.dwWidth      = g_rcScreenRect.right - g_rcScreenRect.left;
ddsd.dwHeight     = g_rcScreenRect.bottom - g_rcScreenRect.top;
memcpy( &ddsd.ddpfPixelFormat, &ddpfZBuffer, sizeof(DDPIXELFORMAT) );

// For hardware devices, the z-buffer should be in video memory. For
// software devices, create the z-buffer in system memory
if( IsEqualIID( *pDeviceGUID, IID_IDirect3DHALDevice ) )
    ddsd.ddsCaps.dwCaps |= DDSCAPS_VIDEOMEMORY;
else
    ddsd.ddsCaps.dwCaps |= DDSCAPS_SYSTEMMEMORY;

// Create and attach a z-buffer. Real apps should be able to handle an
// error here (DDERR_OUTOFVIDEOMEMORY may be encountered). For this
// tutorial, though, we are simply going to exit ungracefully.
if( FAILED( hr = g_pDD->CreateSurface( &ddsd, &g_pddsZBuffer, NULL ) ) )
    return hr;
```

The preceding code simply prepares a **DDSURFACEDESC2** structure for the depth buffer, using the dimensions of the render-target surface calculated from previously set global variables. The pixel format information retrieved during the previous step, Step 1: Enumerate Depth-Buffer Formats, is copied into the surface description.

Note

A hardware device can use a depth buffer regardless of its location in memory. When using a hardware device, it's best to let the device determine the best location for the buffer by omitting the **DDSCAPS_VIDEOMEMORY** and **DDSCAPS_SYSTEMMEMORY** surface capability flags. However, a software device can only be created if the depth buffer exists in system memory. The preceding code checks for this possibility and includes the **DDSCAPS_SYSTEMMEMORY** flag if necessary.

Once the surface description is ready, the code calls the **IDirectDraw7::CreateSurface** method to create the new depth-buffer surface. After the depth buffer is created, it can be attached to the surface that will be used as the render target, as described in Step 3: Attach the Depth Buffer.

Step 3: Attach the Depth Buffer

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

Once the depth buffer is created, you need to attach it to the surface that will be used as the render target. Do this by calling the

IDirectDrawSurface7::AddAttachedSurface method of the render-target surface. The ZBuffer sample performs this with the following code:

```
// Attach the z-buffer to the back buffer.  
if( FAILED( hr = g_pddsBackBuffer->AddAttachedSurface( g_pddsZBuffer ) ) )  
    return hr;
```

Once the depth buffer is attached to the render-target surface, the system will automatically use the depth buffer whenever depth buffering is enabled, as discussed in Step 4: Enable Depth Buffering.

Step 4: Enable Depth Buffering

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

After attaching the depth buffer to the render-target surface, you can create a rendering device from the render target. Given a rendering device, you enable depth buffering by setting the D3DRENDERSTATE_ZENABLE render state for the device. The **D3DZBUFFERTYPE** enumerated type includes members to set the depth-buffer render state. The D3DZB_TRUE member (or TRUE) enables z-buffering. The ZBuffer sample enables z-buffering during scene rendering in the App_Render application-defined function. The following is the appropriate excerpt from App_Render:

```
// Enable z-buffering. (Note: we don't really need to do this every frame.)
pd3dDevice->SetRenderState( D3DRENDERSTATE_ZENABLE, TRUE );
```

Although this tutorial enables depth-buffering each frame, it is not necessary to do so. A real-world application would likely set the D3DRENDERSTATE_ZENABLE render state during scene initialization, only changing to disable depth buffering or to choose another type of depth buffering.

Note

The **D3DZBUFFERTYPE** enumerated type includes the D3DZB_USEW value to enable w-based depth comparisons on compliant hardware. For more information, see Depth Buffers.

Tutorial 3: Using Alternate Primitive Styles

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

The Using Alternate Primitive Styles tutorial demonstrates how to render polygons using the various draw primitive calls provided by Direct3D. The primitives in this tutorial are polygons, which are closed, three-dimensional (3-D) objects identified by at least three vertices.

Specifically, this tutorial demonstrates how to prepare the geometry for a series of primitives (a series of wall segments, a cone, and a cube); it illustrates how to rotate the viewpoint; and it uses matrix concatenation to perform a 3-D transformation: translating, rotating, and scaling a cube.

- Step 1: Initialize the Scene Geometry

- Step 2: Render and Display the Scene

For more information on 3-D primitives, see 3-D Primitives.

Note

The code in the DrawPrims application is similar to the code in the Triangle application. This tutorial focuses only on the geometric preparation and rendering code unique to the DrawPrims application, and does not cover setting up Direct3D, shutting down, or processing system events. For information on these tasks, see Tutorial 1: Rendering a Single Triangle.

The Using Alternate Primitive Styles tutorial uses a depth buffer to store depth information for the scene. To use depth buffering, you must enumerate supported depth-buffer formats, create a depth-buffer surface, attach the depth-buffer surface to a render-target surface, and enable depth buffering for the rendering device. For information on these tasks, see Tutorial 2: Adding a Depth Buffer.

Step 1: Initialize the Scene Geometry

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

This tutorial prepares the geometry for the primitives in the application-defined App_InitDeviceObjects method. App_InitDeviceObjects defines a series of vertices for each object; retaining each primitive's vertex information in globally defined data structures. Vertex information is used by the DrawPrimitive Methods during rendering.

Technically, you are not required to set up the geometry at this time—you can do it anytime prior to calling the rendering methods.

- Step 1.1: Prepare the Wall Segments
- Step 1.2: Prepare the Cone
- Step 1.3: Prepare the Cube

For more information on primitive types, see Primitive Types.

For more information on specifying vertices in Direct3D, see Vertex Formats.

Note

The order in which the geometry is prepared for specific primitives does not effect the layout of objects in your scene.

Step 1.1: Prepare the Wall Segments

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

The Using Alternate Primitive Styles tutorial describes the wall segments in your scene with **D3DVERTEX** structures, which are untransformed and unlit vertices. When you use untransformed and unlit vertices, you effectively request that Direct3D use its own internal algorithms to perform transformation and lighting operations.

You can define each wall segment in 3-D space by calculating and saving the information for a pair of vertices, as demonstrated in the following code from the DrawPrims application:

```
HRESULT App_InitDeviceObjects( LPDIRECT3DDEVICE7 pd3dDevice )
{
    DWORD i;

    for( i=0; i<NUM_WALL_SIDES; i++ )
    {
        FLOAT x = (FLOAT)sin( 2*g_PI*i/(NUM_WALL_SIDES-1) );
        FLOAT z = (FLOAT)cos( 2*g_PI*i/(NUM_WALL_SIDES-1) );

        g_pvWall[2*i+0] = D3DVERTEX( 10.0f * D3DVECTOR( x, -0.1f, z ),
                                     D3DVECTOR( -x, 0, -z ), 0, 0 );
        g_pvWall[2*i+1] = D3DVERTEX( 10.0f * D3DVECTOR( x, 0.1f, z ),
                                     D3DVECTOR( -x, 0, -z ), 0, 0 );
    }
}
```

The preceding code fills an array of **D3DVERTEX** structures, *g_pvWall*., with vertex component data. The code locates the x- and z- coordinates of the vertex by using trigonometric functions. The **D3DVERTEX** structure requires that you specify your coordinates in model coordinates. Since we have used untransformed and unlit vertices, the system will apply world, view, and projection transformations to the model coordinates to properly position them within your scene. Also, the x-, y-, and z-coordinates of the vertex are scaled by a value of ten to properly size the wall segments.

After defining the geometry for the wall segments in the scene, you can move on to preparing another geometric primitive for your scene, a cone. Again, note that the order in which specific objects are defined is not critical.

Defining a cone in 3-D space is demonstrated in Step 1.2: Prepare the Cone.

Step 1.2: Prepare the Cone

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

To prepare the geometry for a cone, you only need to declare a position vector and a normal vector. Therefore, the cone object in the DrawPrims application is defined with the following programmer-defined structure:

```
struct MyFlexibleVertex
{
    D3DVECTOR vPosition;
    D3DVECTOR vNormal;
};
```

You define the tip of the cone with the following code fragment:

```
g_pvCone[0].vPosition = D3DVECTOR( 0, CONE_HEIGHT/2, 0 );
g_pvCone[0].vNormal = Normalize( D3DVECTOR( 0, 1, 0 ) );
```

You compute each side of the cone with the following trigonometric functions:

```
for( i=0; i<NUM_CONE_SIDES; i++ )
{
    FLOAT x = (FLOAT)sin( 2*g_PI*i/(NUM_CONE_SIDES-1) );
    FLOAT y = -CONE_HEIGHT/2;
    FLOAT z = (FLOAT)cos( 2*g_PI*i/(NUM_CONE_SIDES-1) );
    g_pvCone[i+1].vPosition = CONE_RADIUS * D3DVECTOR( x, y, z );
```

You calculate the position vector of each side of the cone with the following code fragment:

```
g_pvCone[i+1].vNormal = Normalize( D3DVECTOR( x, 0.5f, z ) );
}
```

After defining the wall segments and the cone in geometric space, you can move on to preparing the cube. This process is described in Step 1.3: Prepare the Cube.

Step 1.3: Prepare the Cube

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

The cube will be represented as an indexed primitive; thus, the cube will be rendered by indexing into an array of vertices. Specifically, the following indices will be used to index into the cube's vertex list during rendering:

```
WORD* pwIndex = g_pwCubeIndices;
*pwIndex++ = 1; *pwIndex++ = 2; *pwIndex++ = 3;
*pwIndex++ = 2; *pwIndex++ = 1; *pwIndex++ = 0;
*pwIndex++ = 4; *pwIndex++ = 5; *pwIndex++ = 6;
*pwIndex++ = 6; *pwIndex++ = 5; *pwIndex++ = 7;
*pwIndex++ = 3; *pwIndex++ = 2; *pwIndex++ = 6;
*pwIndex++ = 3; *pwIndex++ = 6; *pwIndex++ = 7;
*pwIndex++ = 0; *pwIndex++ = 1; *pwIndex++ = 4;
*pwIndex++ = 4; *pwIndex++ = 1; *pwIndex++ = 5;
*pwIndex++ = 2; *pwIndex++ = 0; *pwIndex++ = 4;
*pwIndex++ = 2; *pwIndex++ = 4; *pwIndex++ = 6;
*pwIndex++ = 1; *pwIndex++ = 3; *pwIndex++ = 5;
*pwIndex++ = 5; *pwIndex++ = 3; *pwIndex++ = 7;
```

After defining the geometry for all of your objects in the scene, you can render and display the scene, which is the topic of Step 2: Render and Display the Scene.

Step 2: Render and Display the Scene

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

To properly display and manipulate your objects in a scene, you need to define the viewpoint of your scene, create the view matrix, perform any necessary transformations, and, of course, render the completed scene. These tasks are explained in the following steps:

- Step 2.1: Define the Viewpoint
- Step 2.2: Create and Set the View Matrix
- Step 2.3: Perform the 3-D Transformation
- Step 2.4: Render the Scene

For more information on rendering primitives, see *Rendering Primitives*.

Step 2.1: Define the Viewpoint

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

This tutorial uses the application-defined subroutine App_FrameMove to rotate the viewpoint around in a circle. The viewpoint is most conveniently defined by three **D3DVECTOR** structures: an eye point, a look-at point, and a vector defining the up direction:

```
HRESULT App_FrameMove( LPDIRECT3DDEVICE7 pd3dDevice, FLOAT fTimeKey )
{
    D3DVECTOR vEyePt  = D3DVECTOR( 5*sinf( fTimeKey), 3, 5*cosf( fTimeKey ) );
    D3DVECTOR vLookatPt = D3DVECTOR( 4*sinf( fTimeKey+0.1f), 2.5, 4*cosf( fTimeKey+0.1f
));
    D3DVECTOR vUpVec   = D3DVECTOR( 0, 1, 0 );
```

You use the three vectors to create a new view matrix:

```
D3DMATRIX matView;
SetViewMatrix( matView, vEyePt, vLookatPt, vUpVec );
```

The preceding code creates a new view matrix for the device by passing the eye point, the look-at point, and the vector defining the up direction to the programmer-defined SetViewMatrix method, which is explained in Step 2.2: Create and Set the View Matrix.

Step 2.2: Create and Set the View Matrix

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Given an eye point, a look-at point, and an up vector, you can manually build a 4x4 view matrix for your application. First, get the z basis vector, which points straight ahead:

```
HRESULT SetViewMatrix( D3DMATRIX& mat, D3DVECTOR& vFrom, D3DVECTOR& vAt,
                      D3DVECTOR& vWorldUp )
{
    // Get the z basis vector, which points straight ahead. This is the
    // difference from the eyepoint to the lookat point.
    D3DVECTOR vView = vAt - vFrom;
```

The z basis vector is the difference from the eye point to the look-at point. You should normalize the z basis vector before using it in your calculations:

```

FLOAT fLength = Magnitude( vView );
if( fLength < 1e-6f )
    return E_INVALIDARG;

```

```

// Normalize the z basis vector
vView /= fLength;

```

The preceding code uses the **Magnitude** function to return the absolute value of the view vector; the **Magnitude** function is available to programmers who define D3D_OVERLOADS in their projects. For more information on the helper functions provided by the D3D_OVERLOADS C++ extensions, see D3D_OVERLOADS Helper Functions.

Now, you can retrieve the dot product and calculate the projection of the z basis vector onto the up vector. The resulting projection is the y basis vector:

```

FLOAT fDotProduct = DotProduct( vWorldUp, vView );

D3DVECTOR vUp = vWorldUp - fDotProduct * vView;

```

Note

DotProduct is a function defined in D3D_OVERLOADS.

The following code fragment normalizes the y-basis vector:

```

vUp /= fLength;

```

To find the x basis vector, determine the cross product of the y and z basis vectors:

```

D3DVECTOR vRight = CrossProduct( vUp, vView );

```

Note

CrossProduct is a function defined in D3D_OVERLOADS.

At this point, you have enough information to begin building the new view matrix. In the following code, the first three rows contain the basis vectors used to rotate the view to point at the look-at point:

```

mat._11 = vRight.x; mat._12 = vUp.x; mat._13 = vView.x; mat._14 = 0.0f;
mat._21 = vRight.y; mat._22 = vUp.y; mat._23 = vView.y; mat._24 = 0.0f;
mat._31 = vRight.z; mat._32 = vUp.z; mat._33 = vView.z; mat._34 = 0.0f;

```

Now, you set the translation values. Note that rotations are still about the eye point:

```

mat._41 = - DotProduct( vFrom, vRight );
mat._42 = - DotProduct( vFrom, vUp );
mat._43 = - DotProduct( vFrom, vView );
mat._44 = 1.0f;

```

After setting the translation values, control returns to the calling application-defined function, App_FrameMove. You can put the new view matrix into effect by calling the **IDirect3DDevice7::SetTransform** method:

```
pd3dDevice->SetTransform( D3DTRANSFORMSTATE_VIEW, &matView );
```

The cube's vertices are transformed by the programmer-defined function, TransformVertices, which is explained in Step 2.3: Perform the 3-D Transformation.

Step 2.3: Perform the 3-D Transformation

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Your scene is nearly completed; however, you still need to manually rotate, transform, and scale the cube's vertices. First, fill the array of **D3DTLVERTEX** structures, *g_pvCube*, with untransformed data for the cube. Note that the first three parameters represent the untransformed position of each vertex in screen coordinates:

```
g_pvCube[0] = D3DTLVERTEX( D3DVECTOR(-1,-1,-1), 0, 0xffffffff, 0, 0, 0 );
g_pvCube[1] = D3DTLVERTEX( D3DVECTOR( 1,-1,-1), 0, 0xff00ffff, 0, 0, 0 );
g_pvCube[2] = D3DTLVERTEX( D3DVECTOR(-1, 1,-1), 0, 0xffff00ff, 0, 0, 0 );
g_pvCube[3] = D3DTLVERTEX( D3DVECTOR( 1, 1,-1), 0, 0xff0000ff, 0, 0, 0 );
g_pvCube[4] = D3DTLVERTEX( D3DVECTOR(-1,-1, 1), 0, 0xffffffff, 0, 0, 0 );
g_pvCube[5] = D3DTLVERTEX( D3DVECTOR( 1,-1, 1), 0, 0xff00ff00, 0, 0, 0 );
g_pvCube[6] = D3DTLVERTEX( D3DVECTOR(-1, 1, 1), 0, 0xffff0000, 0, 0, 0 );
g_pvCube[7] = D3DTLVERTEX( D3DVECTOR( 1, 1, 1), 0, 0xff000000, 0, 0, 0 );
```

Now, call the programmer-defined function TransformVertices to manually transform the cube's eight vertices:

```
TransformVertices( pd3dDevice, g_pvCube, 8 );
```

However, before implementing your 3-D transformation, you will need to retrieve the height and width of the viewport. This information is used to scale the transformed vertices to fit the render window:

```
HRESULT TransformVertices( LPDIRECT3DDEVICE7 pd3dDevice,
                           D3DTLVERTEX* pvVertices, DWORD dwNumVertices )
{
    // Get the width and height of the viewport. This is needed to scale the
    // transformed vertices to fit the render window.
    D3DVIEWPORT7 vp;
    pd3dDevice->GetViewport( &vp );
    DWORD dwClipWidth  = vp.dwWidth/2;
    DWORD dwClipHeight = vp.dwHeight/2;
```

Direct3D uses matrices to perform transformations. In order to execute a 3-D transformation, you need to retrieve the values of the current matrix set. You do this by calling **IDirect3DDevice7::GetTransform** on the rendering device:

```
D3DMATRIX matWorld, matView, matProj;  
pd3dDevice->GetTransform( D3DTRANSFORMSTATE_WORLD,    &matWorld );  
pd3dDevice->GetTransform( D3DTRANSFORMSTATE_VIEW,      &matView );  
pd3dDevice->GetTransform( D3DTRANSFORMSTATE_PROJECTION, &matProj );
```

You use matrix multiplication to concatenate the matrices:

```
D3DMATRIX matTemp = matWorld * matView;  
D3DMATRIX matSet = matTemp * matProj;
```

In the preceding code, a composite matrix, *matSet*, is calculated to represent the product of the desired transformations. During concatenation, ensure that the matrices are multiplied in the order in which you want them to operate.

For more information on matrix concatenation, see [Matrix Concatenation](#).

Now, you transform each vertex through the current matrix set:

```
for( DWORD i=0; i<dwNumVertices; i++ )  
{  
    // Get the untransformed vertex position  
    FLOAT x = pvVertices[i].sx;  
    FLOAT y = pvVertices[i].sy;  
    FLOAT z = pvVertices[i].sz;  
  
    // Transform it through the current matrix set  
    FLOAT xp = matSet._11*x + matSet._21*y + matSet._31*z + matSet._41;  
    FLOAT yp = matSet._12*x + matSet._22*y + matSet._32*z + matSet._42;  
    FLOAT zp = matSet._13*x + matSet._23*y + matSet._33*z + matSet._43;  
    FLOAT wp = matSet._14*x + matSet._24*y + matSet._34*z + matSet._44;
```

Finally, scale the vertices to screen coordinates. In the following code, the first step flattens the coordinates from 3-D space to 2-D device coordinates by dividing each coordinate by the *wp* value. Then, the x- and y-components are transformed from device coordinates to screen coordinates.

```
    pvVertices[i].sx = ( 1.0f + (xp/wp) ) * dwClipWidth;  
    pvVertices[i].sy = ( 1.0f - (yp/wp) ) * dwClipHeight;  
    pvVertices[i].sz = zp / wp;  
    pvVertices[i].rhw = wp;  
}
```

For more information on 3-D transformations, see [3-D Transformations](#).

Note

Device coordinates range from -1 to +1 in the viewport. Also, the *sz*-coordinate will be used in the z-buffer.

Now that you have transformed the cube's eight vertices, you can render a frame of the completed scene. This is shown in [Step 2.4: Render the Scene](#).

Step 2.4: Render the Scene

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

The following **IDirect3DDevice7::DrawPrimitive** call draws a wall segment composed of a **D3DVERTEX**-type triangle strip:

```
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, D3DFVF_VERTEX,
                           g_pvWall, NUM_WALL_VERTICES, NULL );
```

The following **DrawPrimitive** call draws a cone which is a triangle fan of custom, flexible vertices:

```
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, D3DFVF_XYZ|D3DFVF_NORMAL,
                           g_pvCone, NUM_CONE_VERTICES, NULL );
```

Lastly, the following **IDirect3DDevice7::DrawIndexedPrimitive** call draws a cube which is composed of application transformed and lit, indexed vertices:

```
pd3dDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, D3DFVF_TLVERTEX,
                                  g_pvCube, NUM_CUBE_VERTICES,
                                  g_pwCubeIndices, NUM_CUBE_INDICES, NULL );
```

Tutorial 4: Using Device Enumeration

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Device enumeration is useful because it allows applications to query hardware for the Direct3D devices that the hardware supports. As a result, you can build applications that take advantage of particular hardware configurations, or you can let the user choose a particular configuration supported by their system's hardware. The Using Device Enumeration tutorial enumerates the drivers, display modes, and devices on your system and then creates a list of supported display modes for each device.

Basically, device enumeration can be partitioned into two broad steps: (1) initiating and completing device enumeration and (2) selecting an enumerated device.

The Device Enumeration tutorial mirrors this process with the following sections:

- Step 1: Initiate and Complete Device Enumeration
- Step 2: Select an Enumerated Device

For more information on Direct3D devices, see [Direct3D Devices](#).

For more information on device enumeration, see [Enumerating Direct3D Devices](#).

Note

The code in the Enumeration application is similar to the code in the Triangle application. This tutorial focuses only on the device enumeration code unique to the Enumeration application, and does not cover setting up Direct3D, preparing scene geometry, shutting down, or processing system events. For information on these tasks, see [Tutorial 1: Rendering a Single Triangle](#).

The Device Enumeration tutorial uses a depth buffer to store depth information for the scene. To use depth buffering, you must enumerate supported depth-buffer formats, create a depth-buffer surface, attach the depth-buffer surface to a render-target surface, and enable depth buffering for the rendering device. For information on these tasks, see [Tutorial 2: Adding a Depth Buffer](#).

The Device Enumeration tutorial draws a series of cone primitives. For more information on how to draw primitives, see [Tutorial 3: Using Alternate Primitive Styles](#).

Step 1: Initiate and Complete Device Enumeration

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See [Direct3D Immediate Mode Visual Basic Tutorials](#).

[\[C++\]](#)

The Enumeration application uses the following structure to hold information about the enumerated Direct3D devices:

```
struct D3DDEVICEINFO
{
    // D3D Device info
    CHAR        strDesc[40];
    GUID*       pDeviceGUID;
    D3DDEVICEDESC7 ddDeviceDesc;
    BOOL        bHardware;

    // DDraw Driver info
    GUID*       pDriverGUID;
    DDCAPS      ddDriverCaps;
    DDCAPS      ddHELCaps;
```

```
// DDraw Mode Info
DDSURFACEDESC2 ddsdMode;
BOOL      bFullscreen;

// For internal use (apps should not need these)
GUID      guidDevice;
GUID      guidDriver;
DDSURFACEDESC2 ddsdModes[100];
DWORD     dwNumModes;
DWORD     dwCurrentMode;
BOOL      bDesktopCompatible;
};
```

The Enumeration application initiates enumeration with the following call:

```
INT UserDlgSelectDriver( HWND hwndParent )
{
    // Enumerate drivers, devices, and modes
    DirectDrawEnumerateEx( DriverEnumCallback, NULL,
        DDENUM_ATTACHEDSECONDARYDEVICES |
        DDENUM_DETACHEDSECONDARYDEVICES |
        DDENUM_NONDISPLAYDEVICES );
```

In the preceding code, the **DirectDrawEnumerateEx** method is used to enumerate the drivers, devices, and display modes available on the system. The first parameter, *DriverEnumCallback*, is the address of a **DDEnumCallbackEx** function that will be called with a description of each enumerated DirectDraw-enabled HAL.

The process of enumerating devices for your application can be broken down into the following steps:

- Step 1.1: Enumerate Driver Information
- Step 1.2: Enumerate Display Mode Information
- Step 1.3: Enumerate Device Information

The first step for an application that uses device enumeration is to retrieve a reference to the DirectDraw driver. Driver enumeration is explained in Step 1.1: Enumerate Driver Information.

For more information on starting device enumeration, see Starting Device Enumeration.

Step 1.1: Enumerate Driver Information

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

The Enumeration application uses the following application-defined callback function to enumerate all of the DirectDraw devices installed on the system:

```
static BOOL WINAPI DriverEnumCallback( GUID* pGUID, LPSTR strDesc,
                                     LPSTR strName, VOID*, HMONITOR )
```

The first parameter, *pGUID*, is the address of the unique identifier of the DirectDraw object. The next two parameters are strings that describe and name the driver. For more information on this function, consult the **DDEnumCallbackEx** topic in the SDK reference.

Before you can enumerate driver information, you need to create a display driver. You can use the globally-unique identifier (GUID) that is passed into the DriverEnumCallback application-defined callback function to create a display driver for your application:

```
hr = DirectDrawCreateEx( pGUID, (VOID**)&pDD, IID_IDirectDraw7, NULL );
```

The NULL argument indicates that the active display driver should be used. At this point, the Enumeration application creates an **IDirect3D7** object, which will be used to enumerate all of the Direct3D devices.

You can copy the driver data to a **D3DDEVICEINFO** structure, so that driver information can be passed to your application-defined callback functions:

```
// Copy data to a device info structure
ZeroMemory( &d3dDeviceInfo, sizeof(D3DDEVICEINFO) );
strncpy( d3dDeviceInfo.strDesc, strDesc, 39 );
d3dDeviceInfo.ddDriverCaps.dwSize = sizeof(DDCAPS);
d3dDeviceInfo.ddHELCaps.dwSize = sizeof(DDCAPS);
pDD->GetCaps( &d3dDeviceInfo.ddDriverCaps, &d3dDeviceInfo.ddHELCaps );
d3dDeviceInfo.pDriverGUID = pGUID;
```

Consult the following code sample and note that the *ddsModes* member of the **D3DDEVICEINFO** structure is an array of **DDSURFACEDESC2** types that will describe (list) the supported display modes for a particular device. Eventually, you will want to display a combo box to the user that lists the current display mode (windowed) followed by a list of the supported full-screen display modes for each device.

Note

The graphical user interface (GUI) design and structure of the Enumeration application is not explained in this tutorial, refer to the code in the SDK for this information.

The following code sample retrieves the display mode information for the current display mode settings:

```
if( ( NULL == d3dDeviceInfo.pDriverGUID ) &&
    ( d3dDeviceInfo.ddDriverCaps.dwCaps2 & DDAPS2_CANRENDERWINDOWED ) )
{
    // Get the current display depth
```

```

DEVMODE devmode;
devmode.dmSize = sizeof(DEVMODE);
EnumDisplaySettings( NULL, ENUM_CURRENT_SETTINGS, &devmode );

// Set up the mode info
DDSURFACEDESC2* pMode = &d3dDeviceInfo.ddsModes[0];
pMode->ddpfPixelFormat.dwRGBBitCount = devmode.dmBitsPerPel;
pMode->dwWidth = 0;
pMode->dwHeight = 0;

d3dDeviceInfo.dwNumModes = 1;
}

```

The preceding code sample illustrates how to fill the first element of the *ddsModes* array with the current display settings. The remaining elements of the *ddsModes* array will be filled after the full-screen display modes have been enumerated and selected for each device. Note that the **EnumDisplaySettings** method is used to obtain information about the current display depth. The depth settings are then saved in a **DEVMODE** structure, which are then copied into the **ddpfPixelFormat.dwRGBBitCount** member of the first element of the *ddsModes* array:

```
pMode->ddpfPixelFormat.dwRGBBitCount = devmode.dmBitsPerPel;
```

Now, that you have enumerated the DirectDraw drivers on your system and saved the current display settings, you are ready to enumerate display modes.

The details of display mode enumeration are illustrated in Step 1.2: Enumerate Display Mode Information.

Step 1.2: Enumerate Display Mode Information

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

The **IDirectDraw7::EnumDisplayModes** method enumerates all of the display modes that the hardware exposes through the DirectDraw object. The Enumeration application calls the **EnumDisplayModes** function with the following parameters:

```
// Enumerate the fullscreen display modes.
```

```
pDD->EnumDisplayModes( 0, NULL, &d3dDeviceInfo, ModeEnumCallback );
```

The NULL argument specifies that all modes should be enumerated. The third parameter, *&d3dDeviceInfo*, is the address of the **D3DDEVICEINFO** structure containing driver information, which is passed to each enumeration member. The fourth parameter is the address of an **EnumModesCallback2** callback function,

ModeEnumCallback, that the enumeration procedure will call for every mode that the hardware exposes:

```
static HRESULT WINAPI ModeEnumCallback( DDSURFACEDESC2* pddsd,
                                       VOID* pParentInfo )
{
    // Copy the mode into the driver's list of supported modes
    D3DDEVICEINFO* pDriverInfo = (D3DDEVICEINFO*)pParentInfo;
    pDriverInfo->ddsdModes[pDriverInfo->dwNumModes++] = (*pddsd);

    return DDENUMRET_OK;
}
```

In the preceding code, the first parameter, *pddsd*, is a read-only **DDSURFACEDESC2** structure that contains information on the display mode that can be created. The *pParentInfo* parameter (application-defined data representing the display driver) is cast to a **D3DDEVICEINFO** structure, so that it can be safely referenced. After all of the display modes have been enumerated, you are ready to enumerate the Direct3D devices. This step is explained in Step 1.3: Enumerate Device Information.

Step 1.3: Enumerate Device Information

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

The **IDirect3D7::EnumDevices** method enumerates all of the Direct3D device drivers installed on the system. The Enumeration application calls the **EnumDevices** function with the following parameters:

```
pD3D->EnumDevices( DeviceEnumCallback, &d3dDeviceInfo );
```

The first argument, *DeviceEnumCallback*, is the address of a **D3DEnumDevicesCallback7** callback function that will be called every time a Direct3D device is submitted for enumeration. The second parameter is a reference to a **D3DDEVICEINFO** structure, describing the device under enumeration.

For every device that is enumerated, the following function is called:

```
static HRESULT WINAPI DeviceEnumCallback(LPSTR strDesc,
                                       LPSTR strName, D3DDEVICEDESC7* pDesc,
                                       VOID* pParentInfo )
{

```

The first two parameters are the string description and name of the device. The third parameter, *pDesc*, is the address of a **D3DDEVICEDESC7** structure that describes the

hardware capabilities of the Direct3D device. For more information on this function, see the **D3DEnumDevicesCallback7** topic in the SDK reference.

You can set up the device information for the device under enumeration with the following code:

```
D3DDEVICEINFO* pDriverInfo = (D3DDEVICEINFO*)pParentInfo;
D3DDEVICEINFO* pDeviceInfo = &g_d3dDevices[g_dwNumDevices];
memcpy( &pDeviceInfo->ddDeviceDesc, pDesc, sizeof(D3DDEVICEDESC7) );
pDeviceInfo->guidDevice = pDesc->deviceGUID;
pDeviceInfo->pDeviceGUID = &pDeviceInfo->guidDevice;
pDeviceInfo->bHardware = pDesc->dwDevCaps & D3DDEVCAPS_HWRASTERIZATION;
pDeviceInfo->dwNumModes = 0;
pDeviceInfo->ddDriverCaps = pDriverInfo->ddDriverCaps;
pDeviceInfo->ddHELCaps = pDriverInfo->ddHELCaps;
```

In the preceding code, the **D3DDEVICEINFO** structure, *pDeviceInfo*, is filled with the capabilities of the device. Note that *pParentInfo* is cast to a **D3DDEVICEINFO** type, so that the driver GUID and description can be extracted from it.

Now that you have the device information, copy the driver GUID and description for the device into the **D3DDEVICEINFO** structure holding the device information:

```
pDeviceInfo->guidDriver = pDriverInfo->guidDriver;
pDeviceInfo->pDriverGUID = &pDeviceInfo->guidDriver;
strncpy( pDeviceInfo->strDesc, pDriverInfo->strDesc, 39 );
```

After you have enumerated your display driver, display modes, and devices you can select an enumerated device to be used by your application, which is the topic of Step 2: Select an Enumerated Device.

Step 2: Select an Enumerated Device

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

Selecting an enumerated device usually consists of the following steps:

- Step 2.1: Retrieve the Device
- Step 2.2: Test the Device

For more information on selecting enumerated devices, see [Selecting An Enumerated Device](#).

Step 2.1: Retrieve the Device

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Before you can test a device, you must retrieve the device that you want to test. The following section explains how to retrieve device information, so that you can test the device for all of its supported display settings.

```
for( DWORD i=0; i<pDriverInfo->dwNumModes; i++ )
{
    DDSURFACEDESC2 ddsdMode = pDriverInfo->ddsdModes[i];
    DWORD dwModeDepth = ddsdMode.ddpfPixelFormat.dwRGBBitCount;
    DWORD dwBitDepth = pDeviceInfo->ddDeviceDesc.dwDeviceRenderBitDepth;
    BOOL bCompatible = FALSE;
```

The preceding variable assignments retrieve a video mode description for the device, retrieve the bits per pixel support in the returned video mode description, and retrieve the device's rendering bit depth. This information will be used to test the capabilities of the device.

Now that you have the information on the device that you want to test, you can test the device. This step is explained in Step 2.2: Test the Device.

Step 2.2: Test the Device

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

The following code fragment checks the display mode for compatibility with the device, while skipping all 8-bit modes:

```
if( (32==dwModeDepth) && (dwBitDepth&DDBD_32) ) bCompatible = TRUE;
if( (24==dwModeDepth) && (dwBitDepth&DDBD_24) ) bCompatible = TRUE;
if( (16==dwModeDepth) && (dwBitDepth&DDBD_16) ) bCompatible = TRUE;
```

Specifically, the preceding code fragment checks to see if the video mode of the specified device is compatible with the rendering depth of the device under enumeration.

If the device is compatible, it is accepted and copied into the array of device-supported display modes:

```
    if( bCompatible )  
        pDeviceInfo->ddsdModes[pDeviceInfo->dwNumModes++] = ddsdMode;  
}
```

After this loop terminates and all of the devices have been enumerated, the array of display modes, *ddsdModes*, will hold a list of display modes that meet the specified criteria, all other display modes will be ignored.

Tutorial 5: Using Texture Maps

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

This tutorial explains how to draw a simple 3-D scene using textures. In essence, a Direct3D texture is a bitmap of pixel colors; as a result, any bitmap image can be applied to a Direct3D primitive. Textures are a simple, powerful way to add realism to your 3-D objects without altering their geometry. In this way, textures can be used to enhance a primitive with patterns and illusions of color, roughness, transparency, and smoothness. Specifically, the Using Texture Maps tutorial explains how to create textures from file-based bitmaps and how to render a cube primitive with textures.

Note that texture handles are now obsolete in Direct3D; rather, you create and use textures through interface pointers to the texture surfaces. For more information, see Texture Interfaces and Texture Handles.

In order to add textures to the primitives in your 3-D scenes, complete the following steps:

- Step 1: Prepare a Textured Surface
- Step 2: Create a Textured Surface
- Step 3: Render a Textured Primitive

Direct3D supports an extensive texturing feature set, providing developers with easy access to advanced texturing techniques and strategies. For more information on textures and how they can be used and optimized in Direct3D, see Textures.

Note

The code in the Texture application is similar to the code in the Triangle application. This tutorial focuses only on the texturing code unique to the Texture application, and does not cover setting up Direct3D, preparing scene geometry, shutting down, or processing system events. For information on these tasks, see Tutorial 1: Rendering a Single Triangle.

The Texture tutorial draws a cube primitive. For more information on how to draw primitives, see Tutorial 3: Using Alternate Primitive Styles.

The Texture tutorial uses enumeration to enumerate the texture formats. For more information on enumeration techniques and principles, see Tutorial 4: Using Device Enumeration.

Step 1: Prepare a Textured Surface

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

To properly prepare a system-managed texture in a 3-D scene, follow these steps:

- Step 1.1: Set the Texture Coordinates
- Step 1.2: Prepare the Texture Surface
- Step 1.3: Enable Texture Management
- Step 1.4: Enumerate the Texture Formats

Before creating a texture for a primitive, you need to define the vertices of the primitive along its texture coordinates. This task is defined in Step 1.1: Set the Texture Coordinates.

Step 1.1: Set the Texture Coordinates

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Direct3D represents textures as two-dimensional arrays of color values. Each element in the array is referred to as a texel. You can identify a texel through column and row numbers, which are labeled U and V, respectively.

Your application can assign texture coordinates directly to vertices by initializing a **D3DVERTEX** structure with values describing the texture coordinates of the vertex. The following code fragment, taken from the Texture application, creates a set of four vertices describing the front face of a cube:

```
*pVertices++ = D3DVERTEX( D3DVECTOR(-1.0f, 1.0f,-1.0f), n0, 0.0f, 0.0f );
```

```
*pVertices++ = D3DVERTEX( D3DVECTOR( 1.0f, 1.0f,-1.0f), n0, 1.0f, 0.0f );  
*pVertices++ = D3DVERTEX( D3DVECTOR(-1.0f,-1.0f,-1.0f), n0, 0.0f, 1.0f );  
*pVertices++ = D3DVERTEX( D3DVECTOR( 1.0f,-1.0f,-1.0f), n0, 1.0f, 1.0f );
```

For each **D3DVERTEX** assignment, the first argument defines a vertex, the second argument defines the orientation of the vertex normal vector (for the definition of the normal, see the Texture application code in the SDK), and the next two arguments represent the texture coordinates of the vertex. In this case, you want the application to map the entire bitmap texture onto the front face of a cube, so you will use the following texture coordinates: (0, 0), (1, 0), (0, 1), and (1, 1).

For more information on texture coordinates, see Texture Coordinates.

You should employ the same procedures defined previously for the back, top, bottom, right, and left faces of the cube. See the Texture application code in the SDK for information on how to do this.

The Texture application implements the programmer-defined function CreateTexture to return an **IDirectDrawSurface7** object as a texture surface object:

```
g_pddsTexture1 = CreateTexture( pd3dDevice, "tree1.bmp" );  
g_pddsTexture2 = CreateTexture( pd3dDevice, "tex1.bmp" );  
g_pddsTexture3 = CreateTexture( pd3dDevice, "earth.bmp" );
```

In the preceding code, three texture surface objects are created from the following file-based bitmaps: "tree1.bmp", "tex1.bmp", and "earth.bmp." In each function call, the first argument passed, *pd3dDevice*, is a reference to the rendering device for the application. The returned texture surface objects (*g_pddsTexture1*, *g_pddsTexture2*, and *g_pddsTexture3*) will be set to the cube primitive during rendering.

Now, you can look at the programmer-defined function CreateTexture in detail, to see how texture surface objects are prepared and created. The initial step of this process is illustrated in Step 1.2: Prepare the Texture Surface.

Step 1.2: Prepare the Texture Surface

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

In general, before preparing the actual texture surface, you need to create a handle to the bitmap and load the texture file into it. However, in order to create a handle to the bitmap, you need to ensure that your application can find your bitmap file ("tree1.bmp", "tex1.bmp", and "earth.bmp"). As a first step, check the executable's resource for the bitmap file:

```
HBITMAP hbm = (HBITMAP)LoadImage( GetModuleHandle(NULL), strName,  
IMAGE_BITMAP, 0, 0,
```

```
LR_CREATEDIBSECTION);
```

In the preceding **LoadImage** function call, the first argument is a handle to the file used to create the calling process, in other words, the executable's resource. You want this first parameter to return a handle to an instance of the module that contains the image to be loaded. The second parameter identifies the bitmap image to load. The third, fourth, and fifth parameters specify to load a bitmap type and to use the actual width and height of the bitmap image. The final argument, a load flag, loads a bitmap without mapping it to the colors of the display device.

For more information on **LoadImage**, see the Platform SDK.

If the Texture application did not find the file by using the handle to the file used to create the calling process, the Texture application tries to load the bitmap image as a file. Note that a real-world application would try to find the bitmap by searching through multiple file paths.

```
hbm = (HBITMAP)LoadImage( NULL, strName, IMAGE_BITMAP, 0, 0,
LR_LOADFROMFILE|LR_CREATEDIBSECTION );
```

In the preceding **LoadImage** function call, the handle of the instance containing the image is not specified, but the `LR_LOADFROMFILE` load flag is included to indicate that the *strName* parameter is the name of the file that contains the bitmap image.

The work of preparing and creating the texture is done with the following call to the programmer-defined function `CreateTextureFromBitmap`. Note that you pass the function a reference to the rendering device and a handle to the bitmap that you want to use as the texture:

```
return CreateTextureFromBitmap( pd3dDevice, hbm );
```

Note that `CreateTextureFromBitmap` retrieves the device capabilities, so that it can check to see if the device has any constraints when using textures:

```
D3DDEVICEDESC7 ddDesc;
ddDesc.dwSize = sizeof(D3DDEVICEDESC7);
if( FAILED( pd3dDevice->GetCaps( &ddDesc ) ) )
    return NULL;
```

In the preceding code, a **D3DDEVICEDESC7** structure, *ddDesc*, is initialized. Then, the **IDirect3DDevice7::GetCaps** method is used to retrieve a description of the rendering device.

Now, retrieve the bitmap that you want to use as a texture by using the **GetObject** function, so that you can extract the width and height of the bitmap graphics object:

```
BITMAP bm;
GetObject( hbm, sizeof(BITMAP), &bm );
DWORD dwWidth = (DWORD)bm.bmWidth;
DWORD dwHeight = (DWORD)bm.bmHeight;
```

The **BITMAP** structure is a GDI structure that defines a bitmap, for more information see the Platform SDK.

The **GetObject** function is a GDI function that obtains information about a specified graphics object, for more information see the Platform SDK.

You can use a **DDSURFACEDESC2** structure to describe the texture surface. At this point, you can set up the new surface description for the texture:

```
DDSURFACEDESC2 ddsd;
ZeroMemory( &ddsd, sizeof(DDSURFACEDESC2) );
ddsd.dwSize      = sizeof(DDSURFACEDESC2);
ddsd.dwFlags     = DDSD_CAPS|DDSD_HEIGHT|DDSD_WIDTH|
                  DDSD_PIXELFORMAT|DDSD_TEXTURESTAGE;
ddsd.ddsCaps.dwCaps = DDSCAPS_TEXTURE;
ddsd.dwWidth     = dwWidth;
ddsd.dwHeight    = dwHeight;
```

By setting the control flags, specifically **DDSCAPS_TEXTURE**, you inform the system that the surface describes a texture surface. Now, you are ready to enable automatic texture management. This step is explained in Step 1.3: Enable Texture Management.

Step 1.3: Enable Texture Management

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Before creating the texture surface object, you should turn on texture management for the device. Note that you are using the texture manage attribute, so that Direct3D will take care of the complicated details of texture management.

You can turn on texture management for hardware devices with the following code fragment:

```
if( ddDesc.deviceGUID == IID_IDirect3DHALDevice )
    ddsd.ddsCaps.dwCaps2 = DDSCAPS2_TEXTUREMANAGE;
else if( ddDesc.deviceGUID == IID_IDirect3DTnLHalDevice )
    ddsd.ddsCaps.dwCaps2 = DDSCAPS2_TEXTUREMANAGE;
else
    ddsd.ddsCaps.dwCaps |= DDSCAPS_SYSTEMMEMORY;
```

For more information on texture management, see Automatic Texture Management.

Now, use *ddDesc* (a **D3DDEVICEDESC7** structure containing a description of the rendering device) to check device capabilities. You should adjust the width and the height of the surface description, if the driver requires it:

```
if( ddDesc.dpcTriCaps.dwTextureCaps & D3DPTEXTURECAPS_POW2 )
{
```

```

    for( ddsd.dwWidth=1; dwWidth>ddsd.dwWidth; ddsd.dwWidth<=1 );
    for( ddsd.dwHeight=1; dwHeight>ddsd.dwHeight; ddsd.dwHeight<=1 );
}
if( ddDesc.dpcTriCaps.dwTextureCaps & D3DPTTEXTURECAPS_SQUAREONLY )
{
    if( ddsd.dwWidth > ddsd.dwHeight ) ddsd.dwHeight = ddsd.dwWidth;
    else ddsd.dwWidth = ddsd.dwHeight;
}

```

Now that you have enabled texture management and ensured that the surface description is compatible with the driver, you can enumerate the texture formats. This process is defined in Step 1.4: Enumerate the Texture Formats.

Step 1.4: Enumerate the Texture Formats

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Enumerating texture formats gives your application the power to select a supported texture format for the rendering device. The

IDirect3DDevice7::EnumTextureFormats method is used to query the current driver for a list of supported texture formats:

```
pd3dDevice->EnumTextureFormats( TextureSearchCallback, &ddsd.ddpfPixelFormat );
```

In the preceding function call, the *TextureSearchCallback* parameter is the address of the application-defined `D3DEnumPixelFormatsCallback` callback function that will be called for each texture format in the enumeration process.

In the following code excerpt, taken from the `TextureSearchCallback` callback function, the loop is looking for a device-supported texture pixel format. Specifically, the code searches for a 16-bit texture. Note that all other formats, such as alpha textures, are ignored:

```

    if( pddpf->dwFlags & (DDPF_LUMINANCE|DDPF_BUMPLUMINANCE|
DDPF_BUMPDUDV) )
        return DDENUMRET_OK;

    // Skip any FourCC formats
    if( pddpf->dwFourCC != 0 )
        return DDENUMRET_OK;

    // Skip alpha modes
    if( pddpf->dwFlags&DDPF_ALPHAPIXELS )
        return DDENUMRET_OK;

```

```
// We only want 16-bit formats, so skip all others
if( pddpf->dwRGBBitCount != 16 )
    return DDENUMRET_OK;

// We found a good match. Copy the current pixel format to our output
// parameter
memcpy( (DDPIXELFORMAT*)param, pddpf, sizeof(DDPIXELFORMAT) );

// Return with DDENUMRET_CANCEL to end enumeration.
return DDENUMRET_CANCEL;
}
```

Now that a texture format supported by the rendering device has been selected from the enumeration of texture formats, *TextureEnum*, you can create a surface for the texture. This step is detailed in Step 2: Create a Textured Surface.

Step 2: Create a Textured Surface

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

Creating a textured surface from a prepared surface entails the following steps:

- Step 2.1: Retrieve the Render Target
- Step 2.2: Create the New Surface
- Step 2.3: Copy to the Render Surface

Texture surface objects are simply DirectDraw surface object. For more information on texture surfaces, see Texture Surface Objects.

Before creating a textured surface, you need to retrieve the device's render target. This process is explained in Step 2.1: Retrieve the Render Target.

Step 2.1: Retrieve the Render Target

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

At this point, you need to retrieve the device's render target, so that you can use the render target to get a pointer to a DirectDraw object. Then, you need to retrieve the DirectDraw interface, so that you can create a surface.

```
LPDIRECTDRAW7 pddsRender;  
LPDIRECTDRAW7 pDD;  
pd3dDevice->GetRenderTarget( &pddsRender );  
pddsRender->GetDDInterface( (VOID**)&pDD );  
pddsRender->Release();
```

The preceding code uses the **IDirect3DDevice7::GetRenderTarget** method to retrieve a pointer to the DirectDraw surface that is being used as a render target. Then, the **IDirectDrawSurface7::GetDDInterface** method is used to retrieve an interface to the DirectDraw object that was used to create the render target surface.

After retrieving the render target and a pointer to the DirectDraw interface, you can create the surface for your texture. This is explained in Step 2.2: Create the New Surface.

Step 2.2: Create the New Surface

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

Create a new surface for the texture by using the **IDirectDraw7::CreateSurface** method:

```
if( FAILED( hr = pDD->CreateSurface( &ddsd, &pddsTexture, NULL ) ) )  
{  
    pDD->Release();  
    return NULL;  
}
```

The first parameter in the **CreateSurface** method is the address of the **DDSURFACEDESC2** structure describing the surface texture object. After the call completes, the second parameter will be set to a the **IDirectDrawSurface7** interface pointer for your texture surface

The surface that you will use as your texture surface has now been created, you are ready to copy the bitmap to the texture surface. This step is explained in Step 2.3: Copy to the Render Surface.

Step 2.3: Copy to the Render Surface

[Visual Basic]

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[C++]

In order to copy the bitmap to the texture surface, create a device context (DC) for the bitmap and a DC for the surface.

Note

A DC is a GDI structure that stores, retrieves, and modifies the attributes of graphics objects and specifies graphic modes. The DC for the surface contains information that controls the display of graphics on the rendering device. The graphics objects stored in the DC for the bitmap is the bitmap for copying.

You can retrieve the DC for the bitmap with the following code fragment:

```
HDC hdcBitmap = CreateCompatibleDC( NULL );
if( NULL == hdcBitmap )
{
    pddsTexture->Release();
    return NULL;
}
SelectObject( hdcBitmap, hbm );
```

The **CreateCompatibleDC** function is a GDI function that creates a memory DC compatible with the specified device (the application's current screen setting). For more information, see the Platform SDK.

Now, you can retrieve the DC for the surface:

```
HDC hdcTexture;
if( SUCCEEDED( pddsTexture->GetDC( &hdcTexture ) ) )
{
```

In the preceding code, the **IDirectDrawSurface7::GetDC** method returns the DC for the surface.

Complete the process of copying the bitmap image to the surface with the following code:

```
    BitBlt( hdcTexture, 0, 0, bm.bmWidth, bm.bmHeight, hdcBitmap,
            0, 0, SRCCOPY );
    pddsTexture->ReleaseDC( hdcTexture );
}
DeleteDC( hdcBitmap );

// Return the newly created texture
return pddsTexture;
```

```
}
```

The **BitBlt** function is a graphics device interface (GDI) function that performs a bit-block transfer of color data from the specified source DC (*hdcBitmap*) into the destination DC (*hdcTexture*), for more information see the Platform SDK.

Now that you have created a new surface for your texture, you can use that texture during the rendering of the primitives. This task is illustrated in Step 3: Render a Textured Primitive.

Step 3: Render a Textured Primitive

[\[Visual Basic\]](#)

The information in this section pertains only to applications written in C and C++. See Direct3D Immediate Mode Visual Basic Tutorials.

[\[C++\]](#)

The only difference between rendering a textured primitive and a non-textured primitive is that you have to set the texture during the rendering of a textured primitive.

The following code excerpt from the Texture application draws the front and back faces of a cube, using the textured surface *g_pddsTexture1*:

```
pd3dDevice->SetTexture( 0, g_pddsTexture1 );  
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, D3DFVF_VERTEX,  
                           g_pCubeVertices+0, 4, NULL );  
pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, D3DFVF_VERTEX,  
                           g_pCubeVertices+4, 4, NULL );
```

The **IDirect3DDevice7::SetTexture** method assigns a texture to a given stage for the device. Since this tutorial does not use multiple texture blending, the first parameter of **SetTexture** is 0, specifying a value of 0 for the stage identifier. The second parameter identifies the **IDirectDrawSurface7** object being set as the texture.

You should employ the same procedures for the top and bottom faces of the cube, and the left and right faces of the cube, setting *g_pddsTexture2* and *g_pddsTexture3*, respectively. See the Texture application code in the SDK for information on how to do this.

For more information on textured rendering, see Rendering with Texture Surfaces.

For more information on multiple texture blending, see Multiple Texture Blending.

Direct3D Immediate Mode Visual Basic Tutorials

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

The Microsoft® Visual Basic® tutorials in this section show how to create simple Direct3D® applications to do common tasks. These tutorials use many of the Direct3D sample files that are also provided with the DirectX® SDK. The following tutorials are presented here:

- Tutorial 1: Rendering a Single Triangle
 - Tutorial 2: Adding a Depth Buffer
 - Tutorial 3: Using Alternate Primitive Styles
 - Tutorial 4: Using Device Enumeration
 - Tutorial 5: Using Texture Maps
-

Tutorial 1: Rendering a Single Triangle

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

To use Direct3D in Visual Basic, you first create a form for your application window, then you create and initialize a series of DirectDraw® and Direct3D related objects. You manipulate these objects through their defined methods, which are also used to initialize subordinate objects required to render a complete scene.

This tutorial illustrates the steps needed to display the simplest possible scene: a single triangle. After helping you set the properties for your main form, the Triangle tutorial details how to properly initialize DirectDraw and Direct3D, including the rendering of the Direct3DDevice object, the lighting, the viewport, the material, and the model's geometry. Then, the tutorial focuses on a pair of subroutines that demonstrate how to render the triangle frame-by-frame, incrementally rotating the triangle about the y-axis.

In the Triangle tutorial, the following steps are used to set up DirectDraw, to set up Direct3D, and to render a scene:

- Step 1: Create a Form
- Step 2: Initialize System Objects
- Step 3: Initialize the Scene
- Step 4: Execute the Rendering Loop
- Step 5: Shut Down

Note

The Visual Basic code in the Triangle tutorial application separates the DirectDraw and Direct3D initialization code from the application specific code; although this is not necessary, it partitions the code into logical, well-defined units.

Step 1: Create a Form

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

A Visual Basic form serves as your application window. Set the properties for your form in the following manner:

```
BorderStyle = 1 'Fixed Single
Caption     = "Triangle"
ClientHeight = 3195
ClientLeft  = 5415
ClientTop   = 3300
ClientWidth = 3720
LinkTopic   = "Form1"
MaxButton   = 0 'False
MinButton   = 0 'False
ScaleHeight = 213
ScaleMode   = 0 'User
ScaleWidth  = 254
StartPosition = 3 'Windows Default
```

To show your application window, include the following line in your **Form_Load** procedure:

```
Me.Show
```

With the application window ready, you can begin setting up the main DirectX objects, which is the topic of Step 2: Initialize System Objects.

Step 2: Initialize System Objects

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

After you create an application window, you can begin initializing the essential DirectX objects you need to render the scene. For a Direct3D application, this entails creating and configuring DirectDraw, rendering surfaces, a rendering device, and a viewport.

The Triangle tutorial application performs system initialization in the InitDDraw application-defined subroutine, called from the **Form_Load** procedure. The steps executed in the InitDDraw subroutine are presented in the following sub-steps:

- Step 2.1: Initialize DirectDraw
 - Step 2.2: Set Up DirectDraw Surfaces
 - Step 2.3: Initialize Direct3D
 - Step 2.4: Prepare the Viewport
-

Step 2.1: Initialize DirectDraw

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

After creating the application window, the first object that you will create is the DirectDraw object, which is required to set your application's cooperative level. The same DirectDraw object will be used to create both the surface for displaying your scene and the render target for your application's rendering device.

The Triangle tutorial application begins initialization by creating a DirectDraw object, as shown in the following code:

```
Private Sub InitDDraw()  
    ' Create the DirectDraw object and set the application  
    ' cooperative level.  
    Set g_dd = g_dx.DirectDrawCreate("")
```

The preceding code creates a DirectDraw object by calling the **DirectDrawCreate** global function. It passes an empty string to request that the function create a DirectDraw object for the active display driver. For hardware that doesn't support

GDI, such as 3-D only hardware, you should explicitly specify the globally unique identifier (GUID) of the desired driver.

The Triangle tutorial application sets the cooperative level by calling the **DirectDraw7.SetCooperativeLevel** method. Setting the cooperative level tells the system whether or not the application will render in full-screen mode or in a normal window. (Note that some hardware cannot render into a window. You can detect such hardware by checking for the absence of the DDRAW2_CANRENDERWINDOWED capability flag when you call **DirectDraw7.GetCaps**.) The preceding code requests windowed cooperative level, also called the "normal" cooperative level, by including the DDSCL_NORMAL flag in the second parameter passed to **SetCooperativeLevel**. The **SetCooperativeLevel** method can fail if another application already owns full-screen, exclusive mode.

The following code shows how to set the application's cooperative level:

```
g_dd.SetCooperativeLevel Me.hWnd, DDSCL_NORMAL
```

Once you create the DirectDraw object and set the cooperative level, you are ready to prepare the surfaces that will be used to contain and display a rendered scene. The Triangle tutorial does this in Step 2.2: Set Up DirectDraw Surfaces.

Step 2.2: Set Up DirectDraw Surfaces

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

After you create a DirectDraw object and set the cooperative level, you can create the surfaces that your application will use to render and display a scene. Exactly how you create your surface depends largely on whether or not your application will run in a window or in full-screen mode.

Full-screen Application Note

Applications that will run in full-screen mode can create surfaces as shown in the preceding code examples. Most often, these applications should take advantage of page flipping, a feature only available in full-screen, exclusive mode. Using full-screen mode is the topic of a subsequent tutorial. In this case, instead of explicitly creating two surfaces, you can create a flipping chain of surfaces with a single call. For more information, see *Creating Complex Surfaces and Flipping Chains*.

The Triangle tutorial application, designed to run in a window, starts by creating a primary surface that functions as the display:

```
' Prepare and create the primary surface.
```

```
g_ddsd.lFlags = DDSD_CAPS
g_ddsd.ddsCaps.lCaps = DDSCAPS_PRIMARYSURFACE
```

```
Set g_ddPrimary = g_dd.CreateSurface(g_ddsd)
```

The description for the primary surface does not contain information about dimensions or pixel format, these traits are assumed to be the same as the display mode. For example, if the current display mode is 800×600, 16-bit color, DirectDraw ensures that the primary surface matches. After creating the primary surface, you can create the render-target surface. In the case of the tutorial application, the render-target surface is a separate off-screen surface created as follows:

```
' Now create the render-target surface. We are reusing g_ddsd here.
g_ddsd.lFlags = DDSD_HEIGHT Or DDSD_WIDTH Or DDSD_CAPS
g_ddsd.ddsCaps.lCaps = DDSCAPS_OFFSCREENPLAIN Or DDSCAPS_3DDEVICE
```

```
' Use the size of the form to determine the size of the render target
' and viewport rectangle.
g_dx.GetWindowRect Me.hWnd, g_rcDest
```

```
' Set the dimensions of the surface description
g_ddsd.lWidth = g_rcDest.Right - g_rcDest.Left
g_ddsd.lHeight = g_rcDest.Bottom - g_rcDest.Top
```

```
' Create the render-target surface
Set g_ddBackBuffer = g_dd.CreateSurface(g_ddsd)
```

```
' Cache the dimensions of the render target. We'll use
' it for blitting operations.
```

```
With g_rcSrc
    .Left = 0: .Top = 0
    .Bottom = g_ddsd.lHeight
    .Right = g_ddsd.lWidth
End With
```

The preceding code creates an off-screen surface equal to the dimensions of the application window. The off-screen surface is used to cache the image that will later be blitted to the primary surface. There is no need to create a larger surface, because the dimensions of the window dictate what is visible to the user. (This code also initializes a global variable, *g_rcDest*, that is used to set up the viewport and to track the application window size and position.) As the preceding code excerpt shows, you must include the DDSCAPS_3DDEVICE capability for any surface that will be used as a render target. This capability instructs the system to allocate additional internal data structures for 3-D rendering. As when creating the primary surface, the pixel format for the off-screen surface is assumed to be the same as the display mode when it is not provided in the surface description.

Note

Applications that will use a depth buffer should create one and attach it to the render target surface at this point. For simplicity, this tutorial does not employ a depth buffer, but they are covered in Tutorial 2: Adding a Depth Buffer and in Depth Buffers.

After creating the primary and render-target surface, you can create and attach a **DirectDrawClipper** object to the display surface. Using a clipper defines your screen boundaries, freeing you from handling cases when the window is partially obscured by other windows, or when the window is partially outside of the display area. Thus, clippers are not needed for applications that run in full-screen mode. The Triangle tutorial application uses the following code to create a clipper and associate it with the display window:

```
' Create a DirectDrawClipper and attach it to the primary surface.
Dim pcClipper As DirectDrawClipper

Set pcClipper = g_dd.CreateClipper(0)
pcClipper.SetHWND Me.hWnd

g_dd.Primary.SetClipper pcClipper
```

Having created the basic DirectDraw objects, you can move on to setting up the essential Direct3D objects that will be needed to render the scene. The Triangle tutorial demonstrates this task in Step 2.3: Initialize Direct3D.

Step 2.3: Initialize Direct3D

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

After you create the surfaces that your application requires to render and to display a scene, you can begin initializing Direct3D objects by creating a **Direct3D7** object, which is used to create all the other objects you will need to render a scene. You create the **Direct3D7** object by calling **DirectDraw7.GetDirect3D**. The following code from the Triangle tutorial application performs this task:

```
Sub InitD3D()
    Dim d3d As Direct3D7
    Dim ddsd As DDSURFACEDESC2

    ' Retrieve a reference to the Direct3D7 class from the
    ' DirectDraw7 object.
    Set d3d = g_dd.GetDirect3D
```

After creating a **Direct3D7** object, you should create a rendering device to call the **Direct3D7.CreateDevice** method. The **CreateDevice** method accepts the globally unique identifier (GUID) of the desired device and the **DirectDrawSurface7** object, *g_ddsBackBuffer*, that the device will render to. This surface must have been created as a 3-D device by using the DDSCAPS_3DDEVICE capability.

Although the tutorial application uses hard-coded GUID values, a real-world application should enumerate devices to get a GUID. For information about device enumeration, see Enumerating Direct3D Devices.

(The application in the Triangle tutorial checks the display mode prior to creating the device. If the display is set to a palettized mode, the application exits. Attempting to create a device for a palettized surface that does not have an associated palette will cause the **CreateDevice** method to fail. This is done for simplicity. A real-world application should create a render-target surface and attach a palette, or require that the user set their display mode to 16-bit color or higher.)

The following code, taken from the tutorial application, checks the display mode:

```
g_dd.GetDisplayMode ddsd

If ddsd.ddpfPixelFormat.IRGBBitCount <= 8 Then
    MsgBox "This application does not support screen display " & _
        "modes lower than 16-bit."
End
End If
```

Then the tutorial application creates a rendering device:

```
Set g_d3dDevice = d3d.CreateDevice("IID_IDirect3DHALDevice", g_ddsBackBuffer)
```

The call to **Direct3D7.CreateDevice** can fail for many reasons. The most likely cause is when the primary display device does not support 3-D features. Another possibility is if the display hardware cannot render in the current display mode. These possibilities should be checked during device enumeration. To keep the code simple, the Triangle tutorial application attempts to create a software rendering device if the hardware device cannot be created.

Note

Even though the **CreateDevice** method accepts a **DirectDrawSurface7** object, it creates a device object that is separate from a DirectDraw surface object. The device uses a DirectDraw surface as a rendering target.

After the device is created, you can create a viewport object and assign it to the device, as described in Step 2.4: Prepare the Viewport.

Step 2.4: Prepare the Viewport

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

After you create a rendering device, you can create a viewport object and assign it to the rendering device. In short, the viewport determines how the geometry in a 3-D scene is clipped and then represented in the 2-D space of a display screen. For a conceptual overview about viewports, see Viewports and Clipping.

Setting up a viewport is a straight-forward process that starts with preparing the viewport parameters in a **D3DVIEWPORT7** type. The Triangle tutorial application sets the viewport parameters to the dimensions of the render target surface, with a standard range of minimum and maximum depth values into which the scene will be rendered, 0.0 and 1.0, respectively:

```
Dim VPDesc As D3DVIEWPORT7

VPDesc.Width = g_rcDest.Right - g_rcDest.Left
VPDesc.Height = g_rcDest.Bottom - g_rcDest.Top
VPDesc.MinZ = 0#
VPDesc.MaxZ = 1#
```

Once the viewport parameter type is ready, the tutorial application sets the viewport parameters for the rendering device:

```
g_d3dDevice.SetViewport VPDesc
```

Now, cache the viewport parameters in a **RECT** type; you will use this information later during clearing operations.

```
With g_d3dViewport(0)
.X1 = 0: .Y1 = 0
.X2 = VPDesc.Width
.Y2 = VPDesc.Height
End With
```

Now that the basic DirectX objects have been created, you can start preparing the subordinate objects required to render the scene, which is the topic of Step 3: Initialize the Scene.

Step 3: Initialize the Scene

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

After creating the primary Direct3D-related objects—a DirectDraw object, a rendering device, and a viewport—you can begin initializing the scene by preparing the materials, configuring the transformation matrices, and setting up the geometry for your application. The Triangle tutorial application performs the following steps, initializing the scene:

- Step 3.1: Set Up Material and Initial Lighting States
- Step 3.2: Prepare and Set Transformation Matrices
- Step 3.3: Prepare the Geometry

Step 3.1: Set Up Material and Initial Lighting States

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

After you create the basic 3-D rendering objects (a DirectDraw object, a rendering device, and a viewport), you nearly have all that you need to render a simple scene. The next thing to do is to set some initial lighting states and to create and configure a material. If necessary, these settings can all be changed later. For an introduction to these concepts, see *Lighting and Materials*.

The code applies an initial lighting state to the rendering device, *g_d3dDevice*, by calling the **Direct3DDevice7.SetRenderState** method.

```
g_d3dDevice.SetRenderState D3DRENDERSTATE_AMBIENT, _
    g_dx.CreateColorRGBA(1#, 1#, 1#, 1#)
```

In its first parameter, **SetRenderState** passes in the *D3DRENDERSTATE_AMBIENT* flag, associating the call with the ambient light color setting for the scene. The second argument uses the **DirectX7.CreateColorRGBA** method to set the red, green, and blue color values of the ambient light to 1.0, 1.0, and 1.0, respectively. The resulting light emits a white light. Lights do not use the alpha component, which is also set to 1.0 in this case.

Use a **D3DMATERIAL7** type to specify material properties:

```
Dim mtrl As D3DMATERIAL7

mtrl.Ambient.r = 1#: mtrl.Ambient.g = 1#: mtrl.Ambient.b = 0#

' Commit the material to the device.
g_d3dDevice.SetMaterial mtrl
```

When created, a new material has no properties and cannot be used in rendering. The preceding code sets material properties in a **D3DMATERIAL7** type, describing a

material that will reflect the red and green components of ambient light, making the material appear yellow in the scene. (The Triangle tutorial only uses ambient light, so it only sets an ambient reflectance property. A real-world application would use direct light as well as ambient light, and should therefore set diffuse and specular reflectance properties as well.)

Note

When using textures, the object material is usually omitted or colored white.

Now that the material, and initial lighting parameters are set, the Triangle tutorial code sets up the transformation matrices. This is covered in Step 3.2: Prepare and Set Transformation Matrices.

Step 3.2: Prepare and Set Transformation Matrices

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Another step in setting up a simple scene involves setting the world, view, and projection matrices. The system applies these matrices to the model's geometry in order to place the model in the scene, to adjust for the camera's location and orientation, and to scale the vertex data—making distant objects appear smaller than near objects. (For a conceptual overview, see the Geometry Pipeline.)

To create the world matrix, declare a **D3DMATRIX** type, *matWorld*, and create an identity matrix from it. Then, use the **Direct3DDevice7.SetTransform** method with the D3DTRANSFORMSTATE_WORLD flag to define the matrix for the world view.

```
' The world matrix controls the position and orientation of the polygons
' in world space. We'll use it later to spin the triangle.
Dim matWorld As D3DMATRIX
```

```
g_dx.IdentityMatrix matWorld
g_d3dDevice.SetTransform D3DTRANSFORMSTATE_WORLD, matWorld
```

To create the view matrix, declare a **D3DMATRIX** type, *matView*, and create an identity matrix from it. Then, pass the camera coordinates into the **DirectX7.ViewMatrix** method to create the view matrix. Use the **SetTransform** method to define the view matrix for the rendering device.

```
' The view matrix defines the position and orientation of the camera.
' Here, we are just moving it back along the z-axis by 15 units.
Dim matView As D3DMATRIX
```

```
g_dx.IdentityMatrix matView
```

```
Call g_dx.ViewMatrix(matView, MakeVector(0, 0, -15), MakeVector(0, 0, 0), _  
    MakeVector(0, 1, 0), 0)
```

```
g_d3dDevice.SetTransform D3DTRANSFORMSTATE_VIEW, matView
```

The preceding code uses a private, application-defined function, `MakeVector`, to transform three points into a **D3DVECTOR** type:

```
Private Function MakeVector(a As Double, b As Double, c As Double) As D3DVECTOR  
    Dim vecOut As D3DVECTOR
```

```
    With vecOut
```

```
        .x = a
```

```
        .y = b
```

```
        .z = c
```

```
    End With
```

```
    MakeVector = vecOut
```

```
End Function
```

To create the projection matrix, declare a **D3DMATRIX** type, *matProj*, and create an identity matrix from it. Then, pass the clipping planes and the field of view angle into the **DirectX7.ProjectionMatrix** method to create the projection matrix. Use the **SetTransform** method to define the projection matrix for the rendering device.

```
' The projection matrix defines how the 3-D scene is "projected" onto the
```

```
' 2-D render target (the backbuffer surface). Refer to the docs for more
```

```
' info about projection matrices.
```

```
Dim matProj As D3DMATRIX
```

```
g_dx.IdentityMatrix matProj
```

```
Call g_dx.ProjectionMatrix(matProj, 1, 1000, pi / 2)
```

```
g_d3dDevice.SetTransform D3DTRANSFORMSTATE_PROJECTION, matProj
```

In the preceding code, the second parameter passed to **ProjectionMatrix** represents the distance to the near clipping plane in application-specific units, while the third parameter represents the distance to the far clipping plane in application-specific units. The fourth parameter is the field of view angle in radians.

After you have prepared the materials and configured the transformation matrices, you can set up the geometry for your scene. This step is covered in Step 3.3: Prepare the Geometry.

Step 3.3: Prepare the Geometry

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

The Triangle tutorial code initializes the geometry in the application-defined InitGeometry subroutine by defining four **D3DVECTOR** types; a vector type is used for each of the three points used to define the triangle, and for the one point used to define the normal in 3-D space. Technically, you are not required to set up the geometry at this time—you can do it anytime prior to calling the rendering methods:

```
Private Sub InitGeometry()

    ' Set up the geometry for the triangle.
    Dim p1 As D3DVECTOR, _
        p2 As D3DVECTOR, _
        p3 As D3DVECTOR, _
        vNormal As D3DVECTOR

    ' Set values for three points used to define a triangle.
    p1.x = 0#: p1.y = 5#: p1.z = 0#
    p2.x = 5#: p2.y = -5#: p2.z = 0#
    p3.x = -5#: p3.y = -5#: p3.z = 0#

    ' Create a normal vector--shared by the points of the
    ' triangle--that points toward the viewpoint. Note that
    ' we reverse the Z coordinate for the normal of the points
    ' that define the backside of the triangle.
    vNormal.x = 0#: vNormal.y = 0#: vNormal.z = -1#
```

The **DirectX7.CreateD3DVertex** method uses the members of a **D3DVECTOR** type to fill **D3DVERTEX** types by using the individual values that represent discrete vertex components. Fill the global **D3DVERTEX** type array, *g_TriangleVert*, with the triangle's vertex information:

```
    ' Create the 3 vertices for the front of the triangle.
    g_dx.CreateD3DVertex p1.x, p1.y, p1.z, vNormal.x, vNormal.y, vNormal.z, 0, 0, _
        g_TriangleVert(0)
    g_dx.CreateD3DVertex p2.x, p2.y, p2.z, vNormal.x, vNormal.y, vNormal.z, 0, 0, _
        g_TriangleVert(1)
    g_dx.CreateD3DVertex p3.x, p3.y, p3.z, vNormal.x, vNormal.y, vNormal.z, 0, 0, _
        g_TriangleVert(2)

    ' Now do the same for the back of the triangle.
    g_dx.CreateD3DVertex p3.x, p3.y, p3.z, vNormal.x, vNormal.y, -vNormal.z, 0, 0, _
        g_TriangleVert(3)
    g_dx.CreateD3DVertex p2.x, p2.y, p2.z, vNormal.x, vNormal.y, -vNormal.z, 0, 0, _
        g_TriangleVert(4)
```

```
g_dx.CreateD3DVertex p1.x, p1.y, p1.z, vNormal.x, vNormal.y, -vNormal.z, 0, 0, _  
g_TriangleVert(5)
```

The preceding code fragment locates three points in 3-D space that define a triangle standing upright in the $z=0$ plane. After creating all the Direct3D-related objects, initializing the scene, and defining the geometry for your scene, you can render your scene as described in Step 4: Execute the Rendering Loop.

Step 4: Execute the Rendering Loop

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

After you have created the application window, created the DirectX objects, and initialized the scene, you are ready to render the scene. The rendering loop in the Triangle tutorial code renders the scene by calling the application-defined subroutines `RenderScene` and `FrameMove` :

```
Do While g_bRunning = True  
    CNT = CNT + 1  
    RenderScene  
    FrameMove (CNT / 360)  
  
    g_dx.GetWindowRect Me.hWnd, g_rcDest  
  
    j = g_ddsPrimary.Blt(g_rcDest, g_ddsBackBuffer, g_rcSrc, DDBLT_WAIT)  
    If j <> DD_OK Then  
        MsgBox "Couldn't copy the source rectangle to the destination surface." & _  
            Chr$(13) & Hex(j)  
    End  
End If  
DoEvents  
Loop
```

Although the rendering loop appears to be infinite, it terminates when the **Form_Unload** procedure is triggered. For more information, see Step 5: Shut Down.

The rendering loop subdivides the scene rendering task into the following steps:

- Step 4.1: Update the Scene
- Step 4.2: Render the Scene
- Step 4.3: Update the Display
- Step 4.4: Call `DoEvents`

Step 4.1: Update the Scene

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Before a scene is rendered it must be updated. Immediately after `RenderScene` returns, `FrameMove` is called (another application-defined subroutine). `FrameMove` simply updates the world matrix that Direct3D applies to the geometry of the model. The application reflects a rotation around the y-axis based on an internal count value, passed to `FrameMove` in the *stepValue* parameter. Since the rotation is applied once per frame, the end result looks like the model is rotating in place.

```
Private Sub FrameMove(stepVal As Single)
    Dim matSpinY As D3DMATRIX

    Call g_dx.RotateYMatrix(matSpinY, stepVal)
    Call g_d3dDevice.SetTransform(D3DTRANSFORMSTATE_WORLD, matSpinY)
End Sub
```

In the real-world, of course, your applications will do much more than apply a single rotation on a single model. (For more information on rotation matrices, see Rotation in the 3-D Transformations section.)

After you update the model's geometry in the scene, you can render it to the render target surface, as the Triangle tutorial application illustrates in Step 4.2: Render the Scene.

Step 4.2: Render the Scene

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Once the model's geometry has been updated to reflect the desired animation, you can render the scene. The Triangle tutorial code takes a typical approach: the `RenderScene` application-defined subroutine starts by clearing the viewport:

```
Private Sub RenderScene()
    ' Clear the viewport to a blue color.
    g_d3dDevice.Clear 1, g_d3drcViewport(), D3DCLEAR_TARGET, &HFF, 0, 0
```

The preceding code calls the **Direct3DDevice7.Clear** method to clear the viewport. The first two arguments accepted by the **Clear** method are the array of rectangles describing the area or areas on the render target to be cleared and the value that informs the method how many rectangles from the array should be cleared. In most cases, as in the Triangle tutorial application, you will use a single rectangle that covers the entire render target. The third argument determines the method's behavior. **Clear** can clear the render-target surface, the associated depth buffer, the stencil buffer, or any combination of the three. Because this tutorial does not use a depth buffer, **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, respectively. The Triangle tutorial set the clear color for the render target surface to blue. Since the remaining parameters are not used in this tutorial, the code sets them to zero. The **Clear** method will ignore arguments when the corresponding flag is not specified.

After clearing the viewport, the tutorial code informs Direct3D that rendering will begin, renders the scene, then signals that rendering is complete, as shown here:

```
' Begin the scene, render the triangle, then complete the scene
g_d3dDevice.BeginScene
    Call g_d3dDevice.DrawPrimitive(D3DPT_TRIANGLELIST, D3DFVF_VERTEX, _
        g_TriangleVert(0), 6, D3DDP_DEFAULT)
g_d3dDevice.EndScene
```

The **Direct3DDevice7.BeginScene** and **Direct3DDevice7.EndScene** methods notify the system when rendering is beginning or is complete. You can only call rendering methods between calls to these methods. Even if rendering methods fail, you should call **EndScene** before calling **BeginScene** again.

After rendering the scene to the off-screen render target, you can update the user's display. The Triangle tutorial application demonstrates this in Step 4.3: Update the Display.

Step 4.3: Update the Display

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Once a scene has been rendered to the render-target surface, you can show the results on screen. A windowed application usually does this by blitting the content of the render-target surface to the primary surface. A full screen application that employs page-flipping would simply flip the surfaces in the flipping chain. Because it runs in a window, the Triangle tutorial application uses the former method, using the following code:

```
g_dx.GetWindowRect Me.hWnd, g_rcDest
```

```
j = g_ddsPrimary.Blt(g_rcDest, g_ddsBackBuffer, g_rcSrc, DDBLT_WAIT)
```

The preceding **DirectDrawSurface7.Blt** method simply blits the entire contents of the render target surface to the window on the desktop. This tutorial code tracks the destination rectangle for the blit in the *g_rcDest* global variable.

While your application is running, you will need to ensure that control is relinquished to the operating system periodically, which is the topic of Step 4.4: Call DoEvents.

Step 4.4: Call DoEvents

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

You can use the **DoEvents** function to enable the operating system to raise events that apply to your application. In this manner, your application will render individual frames of the scene after the operating system has processed the events in its queue. The Triangle application uses the following line of code to clear the events queue:

```
DoEvents
```

After this call completes, the application returns to the top of the while loop, calling the application-defined subroutines *RenderScene* and *FrameMove*, respectively—rendering a single frame of the scene.

Now that you have your application running, you can go to the final step in the tutorial, Step 5: Shut Down.

Step 5: Shut Down

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

For the Triangle tutorial, the application shut-down code is straight-forward:

```
Private Sub Form_Unload(Cancel As Integer)  
    g_bRunning = False  
End Sub
```

In the preceding code, the **Form_Unload** procedure sets the global variable, *g_bRunning* to false, which terminates the rendering loop.

Tutorial 2: Adding a Depth Buffer

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Direct3D applications often rely on depth buffers to properly display objects in a scene. For a conceptual overview, see Depth Buffers. To use depth buffering you must enumerate supported depth-buffer formats, create a depth-buffer surface, attach the depth-buffer surface to a render-target surface, and enable depth buffering for the rendering device.

The ZBuffer tutorial application performs the following steps to use a z-buffer:

- Step 1: Enumerate Depth-Buffer Formats
- Step 2: Create the Depth Buffer
- Step 3: Attach the Depth Buffer
- Step 4: Enable Depth Buffering

A z-buffer holds the state for every pixel rendered, indicating what depth the rendered pixel is supposed to represent. In this way, when pixels are rendered on top of each other, the rendered pixel can decide if it should be placed behind the pixel that has already been rendered.

Note

The code in the ZBuffer tutorial application is nearly identical to the code in the Triangle tutorial application. This tutorial focuses only on the depth-buffer code unique to ZBuffer, and does not cover setting up Direct3D, rendering, shutting down, or processing events raised by the application. For information on these tasks, see Tutorial 1: Rendering a Single Triangle.

Because some rendering devices require depth buffers to be located in particular places in memory, the system requires that you create and attach the depth-buffer surface to the render-target surface before you create a rendering device.

Step 1: Enumerate Depth-Buffer Formats

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Before you can create a depth buffer, you must determine what depth-buffer formats, if any, are supported by the rendering device. Call the

Direct3D7.GetEnumZBufferFormats method to enumerate the depth-buffer formats that the device supports. The ZBuffer application uses the following code to enumerate depth-buffer formats:

```
' Create the z-buffer after creating the backbuffer and before creating the
' Direct3D device. Note: before creating the z-buffer, applications may want to check
' the device capabilities for the D3DPRASTERCAPS_ZBUFFERLESSHSR flag. This flag is
true
' for certain hardware that can do HSR (hidden-surface removal) without a z-buffer.
' For those devices, there is no need to create a z-buffer.
'

Dim ddpfZBuffer As DDPIXELFORMAT
Dim d3dEnumPFs As Direct3DEnumPixelFormats
```

```
Set g_d3d = g_dd.GetDirect3D
Set d3dEnumPFs = g_d3d.GetEnumZBufferFormats("IID_IDirect3DRGBDevice")
```

The **GetEnumZBufferFormats** method accepts the globally unique identifier (GUID) of the device for which the formats will be enumerated. The ZBuffer application uses a hard-coded GUID; however, a real-world application would implement a testing mechanism to determine the GUID of the rendering device, either IID_IDirect3DRGBDevice or IID_IDirect3DHALDevice.

Now that the formats for the device have been enumerated, you can cycle through each of the supported depth-buffer formats:

```
Dim I As Long

For I = 1 To d3dEnumPFs.GetCount()
    Call d3dEnumPFs.GetItem(I, ddpfZBuffer)
    If ddpfZBuffer.IFlags = DDPF_ZBUFFER Then
        Exit For
    End If
Next I
```

When **Direct3DEnumPixelFormats.GetItem** is called, it passes as its second argument a **DDPIXELFORMAT** type describing the pixel format of the depth buffer. The **IFlags** member will contain **DDPF_ZBUFFER** for any pixel formats that include depth-buffer bits. As its first argument, **GetItem** takes an index into the array of the enumerated formats. If a suitable format is found, **GetItem** saves the pixel format into the *ddpfZBuffer* parameter, a **DDPIXELFORMAT** type, and the loop is terminated.

For simplicity, this tutorial only uses z-buffers, which are the most common type of depth buffer. It ignores any other formats (such as DDPF_STENCILBUFFER) that the system enumerates. Applications could also check the bit depth of the z-buffer (8-, 16-, 24-, 32-bit) and make a choice based on that as well.

After you determine the format of the depth buffer, you can create a `DirectDrawSurface` to use that format, which is the topic of Step 2: Create the Depth Buffer.

Step 2: Create the Depth Buffer

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Now that you have chosen the depth-buffer format, you can create the **DirectDrawSurface** object that will become the depth buffer. The pixel format of the surface is the one determined through enumeration, but the surface dimensions must be identical to the render-target surface to which it will be attached. The ZBuffer application uses the following code for this task:

```
g_ddsd.lFlags = DDSD_CAPS Or DDSD_WIDTH Or DDSD_HEIGHT Or
DDSD_PIXELFORMAT
g_ddsd.ddsCaps.lCaps = DDSCAPS_ZBUFFER
g_ddsd.lWidth = g_rcDest.Right - g_rcDest.Left
g_ddsd.lHeight = g_rcDest.Bottom - g_rcDest.Top
g_ddsd.ddpfPixelFormat = ddpfZBuffer
```

The preceding code simply prepares a **DDSURFACEDESC2** type for the depth buffer, using the dimensions of the render-target surface calculated from previously set global variables. Finally, the pixel format information retrieved during the previous step, Step 1: Enumerate Depth-Buffer Formats, is copied into the surface description:

```
g_ddsd.ddpfPixelFormat = ddpfZBuffer
```

Note

A hardware device can use a depth buffer regardless of its location in memory. When using a hardware device, it is best to let the device determine the best location for the buffer by omitting the `DDSCAPS_VIDEOMEMORY` and `DDSCAPS_SYSTEMMEMORY` surface capability flags. However, a software device can only be created if the depth buffer exists in system memory. In the following code, the ZBuffer tutorial application sets the `DDSCAPS_SYSTEMMEMORY` flag; although, your application should implement a simple searching mechanism to determine whether your application should allocate the surface memory in system memory or in display memory.

' Specify DDSCAPS_VIDEOMEMORY if your surface exists in display memory.

' See the SDK documentation for more information.

`g_ddsdd.ddsCaps.lCaps = g_ddsdd.ddsCaps.lCaps Or DDSCAPS_SYSTEMMEMORY`

Once the surface description is ready, the code calls the **DirectDraw7.CreateSurface** method to create the new depth-buffer surface:

`Set g_ddsZBuffer = g_dd.CreateSurface(g_ddsdd)`

After the depth buffer is created, it can be attached to the surface that will be used as the render target, as described in Step 3: Attach the Depth Buffer.

Step 3: Attach the Depth Buffer

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Once the depth buffer is created, you need to attach it to the surface that will be used as the render target. Do this by calling the

DirectDrawSurface7.AddAttachedSurface method of the render-target surface. The ZBuffer application performs this with the following code:

' Attach the z-buffer surface to the back buffer surface.

`g_ddsBackBuffer.AddAttachedSurface g_ddsZBuffer`

Once the depth buffer is attached to the render-target surface, the system will automatically use the depth buffer whenever depth buffering is enabled, as discussed in Step 4: Enable Depth Buffering.

Step 4: Enable Depth Buffering

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

After attaching the depth buffer to the render-target surface, you can create a rendering device from the render target. Given a rendering device, you enable depth buffering by setting the `D3DRENDERSTATE_ZENABLE` render state for the device. The **CONST_D3DZBUFFERTYPE** enumerated type includes members to set the depth-buffering render state. The `D3DZB_TRUE` member (or `True`), enables

z-buffering. The ZBuffer tutorial application enables z-buffering during scene rendering in the RenderScene application-defined subroutine. The following is the appropriate excerpt from RenderScene:

```
g_d3dDevice.SetRenderState D3DRENDERSTATE_ZENABLE, D3DZB_TRUE
```

Although this tutorial enables depth-buffering for each frame, it is not necessary to do so. A real-world application would likely set the D3DRENDERSTATE_ZENABLE render state during scene initialization, only changing to disable depth buffering or to choose another type of depth buffering.

Note

The **CONST_D3DZBUFFERTYPE** enumerated type includes the D3DZB_USEW value to enable w-based depth comparisons on compliant hardware. For more information, see Depth Buffers.

Tutorial 3: Using Alternate Primitive Styles

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

The Using Alternate Primitive Styles tutorial demonstrates how to render polygons using the various draw primitive calls provided by Direct3D for Visual Basic. The primitives in this tutorial are polygons, which are closed, three-dimensional (3-D) objects identified by at least three vertices. The tutorial uses the DrawPrims application.

Specifically, this tutorial demonstrates how to prepare the geometry for a series of primitives (a series of wall segments, a cone, and a cube), how to rotate the viewpoint, and how to use matrix concatenation to perform a 3-D transformation: translating, rotating, and scaling a cube.

- Step 1: Initialize the Scene Geometry
- Step 2: Render and Display the Scene

For more information on 3-D primitives, see 3-D Primitives.

Note

The code in the DrawPrims application is similar to the code in the Triangle application. This tutorial focuses only on the geometric preparation and rendering code unique to the DrawPrims application, and does not cover setting up Direct3D, shutting down, or processing system events. For information on these tasks, see Tutorial 1: Rendering a Single Triangle.

The Using Alternate Primitive Styles tutorial uses a depth buffer to store depth information for the scene. To use depth buffering, you must enumerate supported depth-buffer formats, create a depth-buffer surface, attach the depth-buffer surface to a render-target surface, and enable depth buffering for the rendering device. For information on these tasks, see Tutorial 2: Adding a Depth Buffer.

Step 1: Initialize the Scene Geometry

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

This tutorial prepares the geometry for the primitives in the application-defined InitGeometry subroutine. InitGeometry defines a series of vertices for each object; retaining each primitive's vertex information in globally defined data structures. Vertex information is used by the DrawPrimitive Methods during rendering.

Technically, you are not required to set up the geometry at this time—you can do it anytime prior to calling the rendering methods.

- Step 1.1: Prepare the Wall Segments
- Step 1.2: Prepare the Cone
- Step 1.3: Prepare the Cube

For more information on primitive types, see Primitive Types.

For more information on specifying vertices in Direct3D, see Vertex Formats.

Note

The order in which the geometry is prepared for specific primitives does not effect the layout of objects in your scene.

Step 1.1: Prepare the Wall Segments

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

The Using Alternate Primitive Styles tutorial describes the wall segments in your scene with **D3DVERTEX** types, which are untransformed and unlit vertices. When you use untransformed and unlit vertices, you effectively request that Direct3D use its own internal algorithms to perform transformation and lighting operations.

You can define each wall segment in 3-D space by calculating and saving the information for a pair of vertices, as demonstrated in the following code from the DrawPrims application:

```
Private Sub InitGeometry()

    ' Set up the wall geometry.
    Dim x As Double, _
        y As Double, _
        z As Double

    Dim i As Integer

    For i = 0 To NUM_WALL_SIDES - 1
        x = Sin(2 * pi * i / (NUM_WALL_SIDES - 1))
        z = Cos(2 * pi * i / (NUM_WALL_SIDES - 1))

        Call g_dx.CreateD3DVertex(x * 10#, -0.1 * 10#, z * 10#, -x, 0, -z, 0, 0, g_vWall(2 * i + 0))
        Call g_dx.CreateD3DVertex(x * 10#, 0.1 * 10#, z * 10#, -x, 0, -z, 0, 0, g_vWall(2 * i + 1))

    Next i
```

In the preceding code, the **DirectX7.CreateD3DVertex** method is used to fill an array of **D3DVERTEX** types, *g_vWall*, with vertex component data. The code locates the x- and z- coordinates of the vertex by using trigonometric functions. The **CreateD3DVertex** method requires that you specify your coordinates in model coordinates. Since we have used untransformed and unlit vertices, the system will apply world, view, and projection transformations to the model coordinates to properly position them within your scene. Also, the x-, y-, and z-coordinates of the vertex are scaled by a value of ten to properly size the wall segments.

After defining the geometry for the wall segments in the scene, you can move on to preparing another geometric primitive for your scene, a cone. Again, note that the order in which specific objects are defined is not critical.

Defining a cone in 3-D space is demonstrated in Step 1.2: Prepare the Cone.

Step 1.2: Prepare the Cone

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

To prepare the geometry for a cone, you only need to declare a position vector and a normal vector. Therefore, the cone object in the DrawPrims application is defined with the following programmer-defined type:

```
Private Type MyFlexibleVertex
    vPosition As D3DVECTOR
    vNormal As D3DVECTOR
End Type
```

You can define the tip of the cone with the following code fragment:

```
Dim vectorNorm As D3DVECTOR

g_vCone(0).vPosition = MakeVector(0, CONE_HEIGHT / 2, 0)

vectorNorm = MakeVector(0, 1, 0)
Call g_dx.vectorNormalize(vectorNorm)
g_vCone(0).vNormal = vectorNorm
```

You can compute each side of the cone with the following trigonometric functions:

```
For i = 0 To NUM_CONE_SIDES - 1
    x = Sin(2 * pi * i / (NUM_CONE_SIDES - 1))
    y = -CONE_HEIGHT / 2
    z = Cos(2 * pi * i / (NUM_CONE_SIDES - 1))

    g_vCone(i + 1).vPosition = MakeVector(x * CONE_RADIUS, y * CONE_RADIUS, z *
CONE_RADIUS)
```

You can calculate the position vector of each side of the cone with the following code fragment:

```
vectorNorm = MakeVector(x, 0.5, z)
Call g_dx.vectorNormalize(vectorNorm)
g_vCone(i + 1).vNormal = vectorNorm
Next i
```

After defining the wall segments and the cone in geometric space, you can move on to preparing the cube. This process is described in Step 1.3: Prepare the Cube.

Step 1.3: Prepare the Cube

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

The cube will be represented as an indexed primitive; thus, the cube will be rendered by indexing into an array of vertices. Specifically, the following indices will be used to index into the cube's vertex list during rendering:

```
g_nCubeIndices(0) = 1: g_nCubeIndices(1) = 2: g_nCubeIndices(2) = 3
g_nCubeIndices(3) = 2: g_nCubeIndices(4) = 1: g_nCubeIndices(5) = 0
g_nCubeIndices(6) = 4: g_nCubeIndices(7) = 5: g_nCubeIndices(8) = 6
g_nCubeIndices(9) = 6: g_nCubeIndices(10) = 5: g_nCubeIndices(11) = 7
g_nCubeIndices(12) = 3: g_nCubeIndices(13) = 2: g_nCubeIndices(14) = 6
g_nCubeIndices(15) = 3: g_nCubeIndices(16) = 6: g_nCubeIndices(17) = 7
g_nCubeIndices(18) = 0: g_nCubeIndices(19) = 1: g_nCubeIndices(20) = 4
g_nCubeIndices(21) = 4: g_nCubeIndices(22) = 1: g_nCubeIndices(23) = 5
g_nCubeIndices(24) = 2: g_nCubeIndices(25) = 0: g_nCubeIndices(26) = 4
g_nCubeIndices(27) = 2: g_nCubeIndices(28) = 4: g_nCubeIndices(29) = 6
g_nCubeIndices(30) = 1: g_nCubeIndices(31) = 3: g_nCubeIndices(32) = 5
g_nCubeIndices(33) = 5: g_nCubeIndices(34) = 3: g_nCubeIndices(35) = 7
```

After defining the geometry for all of your objects in the scene, you can render and display the scene, which is the topic of Step 2: Render and Display the Scene.

Step 2: Render and Display the Scene

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

To properly display and manipulate your objects in a scene, you need to update the view of your scene, perform any necessary transformations, and, of course, render the completed scene. These tasks are explained in the following steps:

- Step 2.1: Update the Scene
- Step 2.2: Perform the 3-D Transformation
- Step 2.3: Render the Scene

For more information on rendering primitives, see *Rendering Primitives*.

Step 2.1: Update the Scene

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

This tutorial uses the application-defined subroutine `FrameMove` to rotate the viewpoint around in a circle. The viewpoint is most conveniently defined by three **D3DVECTOR** types: an eye point, a look-at point, and a vector defining the up direction, as shown in the following code:

```
Dim vEyePt As D3DVECTOR, _
    vLookatPt As D3DVECTOR, _
    vUpVec As D3DVECTOR

vEyePt = MakeVector(5 * Sin(stepVal), 3, 5 * Cos(stepVal))
vLookatPt = MakeVector(4 * Sin(stepVal + 0.1), 2.5, 4 * Cos(stepVal + 0.1))
vUpVec = MakeVector(0, 1, 0)
```

Now, you can use the three vectors to create and set a new view matrix:

```
Dim matView As D3DMATRIX

Call g_dx.ViewMatrix(matView, vEyePt, vLookatPt, vUpVec, stepVal / 360)
```

```
g_d3dDevice.SetTransform D3DTRANSFORMSTATE_VIEW, matView
```

The preceding code creates a new view matrix for each rendered frame of the scene by passing the new location of the camera to the **DirectX7.ViewMatrix** method. Next the **Direct3DDevice7.SetTransform** method sets the newly-generated view matrix for the device. The value of *stepValue* (passed into the application-defined method, `FrameMove`) rotates the viewpoint one degree at a time. As a result, the wall segments and the cone appear to rotate 360 degrees.

At this point, you have nearly completed your scene; however, you still need to manually rotate, transform, and scale the cube's vertices. First, use the **DirectX7.CreateD3DTLVertex** method to fill the array of **D3DTLVERTEX** types with untransformed data for the cube. Note that the first three parameters of **DirectX7.CreateD3DTLVertex** represent the untransformed position of each vertex in screen coordinates:

```
g_dx.CreateD3DTLVertex -1, -1, -1, 0, &HFFFFFF, 0, 0, 0, g_vCube(0)
g_dx.CreateD3DTLVertex 1, -1, -1, 0, &HFF&, 0, 0, 0, g_vCube(1)
g_dx.CreateD3DTLVertex -1, 1, -1, 0, &HFF00&, 0, 0, 0, g_vCube(2)
g_dx.CreateD3DTLVertex 1, 1, -1, 0, &HFF0000, 0, 0, 0, g_vCube(3)
g_dx.CreateD3DTLVertex -1, -1, 1, 0, &HFFFF&, 0, 0, 0, g_vCube(4)
g_dx.CreateD3DTLVertex 1, -1, 1, 0, &HFF0066, 0, 0, 0, g_vCube(5)
g_dx.CreateD3DTLVertex -1, 1, 1, 0, &H9966FF, 0, 0, 0, g_vCube(6)
g_dx.CreateD3DTLVertex 1, 1, 1, 0, &HFF00FF, 0, 0, 0, g_vCube(7)
```

Now, you can let the application transform the cube's eight vertices:

```
Call TransformVertices(g_d3dDevice, g_vCube, NUM_CUBE_VERTICES)
```

The cube's vertices are transformed by the application-defined function, `TransformVertices`, which is explained in Step 2.2: Perform the 3-D Transformation.

Step 2.2: Perform the 3-D Transformation

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

The Using Alternate Primitive Styles tutorial uses geometric transformation to define the location of the cube relative to the other objects in the scene, to rotate the cube, and to scale the cube in the scene.

Before implementing your 3-D transformation, you will need to retrieve the height and width of the viewport. This information is used to scale the transformed vertices to fit the render window:

```
Private Sub TransformVertices(d3dDevice As Direct3DDevice7, vVertices() As D3DTLVERTEX,
    nNumVertices As Integer)
```

```
    Dim vp As D3DVIEWPORT7
    Dim nClipWidth As Integer
    Dim nClipHeight As Integer
```

```
    ' Get the height and width of the viewport. This is needed to scale the transformed vertices
    ' to fit the render window.
    d3dDevice.GetViewport vp
    nClipWidth = vp.lWidth / 2
    nClipHeight = vp.lHeight / 2
```

Direct3D uses matrices to perform transformations. In order to execute a 3-D transformation, you need to retrieve the values of the current matrix set. You do this by calling **Direct3DDevice7.GetTransform** on the rendering device:

```
d3dDevice.GetTransform D3DTRANSFORMSTATE_WORLD, matWorld
d3dDevice.GetTransform D3DTRANSFORMSTATE_VIEW, matView
d3dDevice.GetTransform D3DTRANSFORMSTATE_PROJECTION, matProj
```

You use matrix multiplication to concatenate the matrices:

```
Dim matSet As D3DMATRIX

g_dx.IdentityMatrix matSet

' Concatenate the matrices.
g_dx.MatrixMultiply matSet, matWorld, matView
g_dx.MatrixMultiply matSet, matSet, matProj
```

In the preceding code, a composite matrix, *matSet*, is calculated to representing the product of the desired transformations. During concatenation, ensure that the matrices are multiplied in the order in which you want them to operate.

For more information on matrix concatenation, see Matrix Concatenation.

Now, you transform each vertex through the current matrix set:

```
For i = 0 To nNumVertices - 1
    ' Get the untransformed vertex position.
    x = vVertices(i).sx
    y = vVertices(i).sy
    z = vVertices(i).sz

    xp = matSet.rc11 * x + matSet.rc21 * y + matSet.rc31 * z + matSet.rc41
    yp = matSet.rc12 * x + matSet.rc22 * y + matSet.rc32 * z + matSet.rc42
    zp = matSet.rc13 * x + matSet.rc23 * y + matSet.rc33 * z + matSet.rc43
    wp = matSet.rc14 * x + matSet.rc24 * y + matSet.rc34 * z + matSet.rc44
```

Finally, scale the vertices to screen coordinates. In the following code, the first step flattens the coordinates from 3-D space to 2-D device coordinates by dividing each coordinate by the *wp* value. Then, the x- and y-components are transformed from device coordinates to screen coordinates.

```
vVertices(i).sx = (1# + (xp / wp)) * nClipWidth
vVertices(i).sy = (1# - (yp / wp)) * nClipHeight
vVertices(i).sz = zp / wp
vVertices(i).rhw = wp
Next i
```

For more information on 3-D transformations, see 3-D Transformations.

Note

Device coordinates range from -1 to +1 in the viewport. Also, the *sz*-coordinate will be used in the z-buffer.

Now that you have transformed the cube's eight vertices, you can render a frame of the completed scene. This is shown in Step 2.3: Render the Scene.

Step 2.3: Render the Scene

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

The following **Direct3DDevice7.DrawPrimitive** call draws a wall segment composed of a **D3DVERTEX** type triangle strip:

```
Call g_d3dDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, D3DFVF_VERTEX, g_vWall(0),
    NUM_WALL_VERTICES, D3DDP_DEFAULT)
```

The following **DrawPrimitive** call draws a cone which is a triangle fan of custom, flexible vertices:

```
Call g_d3dDevice.DrawPrimitive(D3DPT_TRIANGLEFAN, D3DFVF_XYZ Or  
D3DFVF_NORMAL, g_vCone(0), _  
NUM_CONE_VERTICES, D3DDP_DEFAULT)
```

Lastly, the following **Direct3DDevice7.DrawIndexedPrimitive** call draws a cube which is composed of application transformed and lit, indexed vertices:

```
Call g_d3dDevice.DrawIndexedPrimitive(D3DPT_TRIANGLELIST, D3DFVF_TLVERTEX,  
g_vCube(0), _  
NUM_CUBE_VERTICES, g_nCubeIndices, NUM_CUBE_INDICES, D3DDP_DEFAULT)
```

Tutorial 4: Using Device Enumeration

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Device enumeration is useful because it allows applications to query hardware for the Direct3D devices that the hardware supports. As a result, you can build applications that take advantage of particular hardware configurations, or you can let the user choose a particular configuration supported by their system's hardware. The Using Device Enumeration tutorial enumerates the drivers, display modes, and devices on your system and then creates a list of supported display modes for each device.

Basically, device enumeration can be partitioned into two broad steps: (1) initiating and completing device enumeration and (2) selecting an enumerated device.

The Using Device Enumeration tutorial mirrors this process with the following sections:

- Step 1: Initiate and Complete Device Enumeration
- Step 2: Select an Enumerated Device

For more information on Direct3D devices, see Direct3D Devices.

For more information on device enumeration, see Enumerating Direct3D Devices.

Note

The code in the Enumeration application is similar to the code in the Triangle application. This tutorial focuses only on the device enumeration code unique to the Enumeration application, and does not cover setting up Direct3D, preparing scene geometry, shutting down, or processing system events. For information on these tasks, see Tutorial 1: Rendering a Single Triangle.

The Using Device Enumeration tutorial uses a depth buffer to store depth information for the scene. To use depth buffering, you must enumerate supported depth-buffer formats, create a depth-buffer surface, attach the depth-buffer surface to a render-target surface, and enable depth buffering for the rendering device. For information on these tasks, see Tutorial 2: Adding a Depth Buffer.

The Using Device Enumeration tutorial draws a series of cone primitives. For more information on how to draw primitives, see Tutorial 3: Using_Alternate Primitive Styles.

Step 1: Initiate and Complete Device Enumeration

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

A Visual Basic application initiates enumeration by retrieving a reference to an enumerator class. For the Using Device Enumeration tutorial, you need to declare references to the following enumerator classes:

- **DirectDrawEnum**
- **DirectDrawEnumModes**
- **Direct3DEnumDevices**

The process of enumerating devices for your application can be broken down into the following steps:

- Step 1.1: Enumerate Driver Information
- Step 1.2: Enumerate Display Mode Information
- Step 1.3: Enumerate Device Information

The first step for an application that uses device enumeration is to retrieve a reference to a DirectDraw driver. The Enumeration sample uses the following application-defined subroutine call to enumerate the drivers that can be used by DirectDraw:

```
' Get driver information.  
EnumDriver
```

Driver enumeration is explained in Step 1.1: Enumerate Driver Information.

For more information on starting device enumeration, see Starting Device Enumeration.

Step 1.1: Enumerate Driver Information

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Create a display driver for your application by calling **DirectX7.DirectDrawCreate** on the global **DirectX7** object:

```
Set g_dd = g_dx.DirectDrawCreate("")
```

Now, you can retrieve driver information. For this simple tutorial we are just using the primary display driver:

```
Set g_ddEnum = g_dx.GetDDEnum()
```

```
For i = 1 To g_ddEnum.GetCount()
```

```
    g_sDriverGUID = g_ddEnum.GetDescription(i)
```

```
Next i
```

The preceding code fragment uses the **DirectX7.GetDDEnum** method to create a **DirectDrawEnum** object. Then, the **DirectDrawEnum** object, *g_ddEnum*, is used to enumerate the video driver display adapters installed on the system. When the primary display driver is found, **DirectDrawEnum.GetDescription** is used to return the driver description of the specified DirectDraw device.

The Enumeration sample application uses the following application-defined subroutine call to perform display mode enumeration:

```
' Enumerate the display modes.
```

```
EnumModes
```

The details of display mode enumeration are illustrated in Step 1.2: Enumerate Display Mode Information.

Step 1.2: Enumerate Display Mode Information

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

You can use the **DirectDraw7.GetDisplayModesEnum** method to return a reference to a **DirectDrawEnumModes** class. The retrieved display mode information is contained in a **DDSURFACEDESC2** type, *g_enumInfo*:

```
Set g_ddEnumModes = g_dd.GetDisplayModesEnum(DDEDM_DEFAULT, g_enumInfo)
```

In the preceding code the `DDEDM_DEFAULT` flag tells the application to perform a normal enumeration of the display modes. For more information on available options, see the **CONST_DDEDMFLAGS** topic in the SDK reference. Now that you have enumerated the DirectDraw driver and the display modes, you can move on to enumerating the devices. The Enumeration application uses the following application-defined subroutine to enumerate the devices:

```
' Enumerate the devices.
EnumDevices
```

Device enumeration is explained in Step 1.3: Enumerate Device Information.

Step 1.3: Enumerate Device Information

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Before you can begin device enumeration, you need to retrieve the Direct3D object for the current display device:

```
Set g_d3d = g_dd.GetDirect3D()
```

Now, retrieve a reference to the **Direct3DEnumDevices** class by calling the **Direct3D7.GetDevicesEnum** method:

```
Set g_d3dEnumDevices = g_d3d.GetDevicesEnum()
For i = 1 To g_d3dEnumDevices.GetCount()
    cmbDevice.AddItem g_d3dEnumDevices.GetName(i)
Next
```

The preceding code sample places the user-friendly name of each enumerated device into a combo box, *cmbDevice*. After you have a reference to the **Direct3DEnumDevices** class, you can call methods that return the number of enumerated devices and information about each device:

Call the **Direct3DEnumDevices.GetCount** method to return the number of enumerated devices. Note that the first device is at index 1, not index 0.

Call the **Direct3DEnumDevices.GetGuid** method to return a text string representation of the globally-unique identifier (GUID) for an enumerated device.

Call the **Direct3DEnumDevices.GetName** and **Direct3DEnumDevices.GetDescription** methods to return text strings that contain the name and user-friendly description of the device.

Call the **Direct3DEnumDevices.GetDesc** method to retrieve a description of the capabilities for the device, in the form of a **D3DDEVICEDESC7** type.

After you have enumerated your display driver, display modes, and devices you can select an enumerated device to be used by your application, which is the topic of Step 2: Select an Enumerated Device.

Step 2: Select an Enumerated Device

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Selecting an enumerated device usually consists of the following steps:

- Step 2.1: Retrieve the Device
- Step 2.2: Test the Device

The Enumeration application uses the following application-defined subroutine call to select an enumerated device and build a list of display modes that are supported by that device:

```
' Select an enumerated device and build a list of supported display modes  
' for that device.  
SelectDisplayModes
```

The process of retrieving an enumerated device is defined in Step 2.1: Retrieve the Device.

For more information on selecting enumerated devices, see Select an Enumerated Device.

Step 2.1: Retrieve the Device

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Before you can test a device, you must retrieve the device that you want to test. The following section explains how to retrieve the current display settings and how to retrieve a device reference, so that you can test the device for all of its supported display settings.

The Enumeration sample uses the **DirectDraw7.GetDisplayMode** method to retrieve the current display mode information in a **DDSURFACEDESC2** type:

```
g_dd.GetDisplayMode desc
currMode = desc.ddpfPixelFormat.IRGBBitCount
```

The preceding code saves the bits per pixel setting of the current display in *currMode*. Then, the Enumeration sample adds the current display mode to a combo box, *cmbMode*, that is displayed to the user:

```
cmbMode.Clear
cmbMode.AddItem Str(currMode) + "-bit Display Window"
```

Now, to retrieve the device that you want to test, determine which device has been selected by the user in the combo box, *cmbDevice*, by using the

Direct3DEnumDevices.GetDesc method:

```
Call g_d3dEnumDevices.GetDesc(cmbDevice.ListIndex + 1, deviceDesc)
```

In the preceding code, a **DDSURFACEDESC2** type, *deviceDesc*, is used to hold surface description information for the device that you want to test. This is the device that you will use to build a list of supported display modes. Ultimately, you will place these supported display modes into a combo box, so that the user can choose the display mode for the Enumeration application.

Note

The design and structure of the graphical user interface (GUI) of the Enumeration application is not discussed in this tutorial. To gain an understanding of how the code samples in the Using Device Enumeration tutorial interact with the GUI, you should consult the complete Enumeration application code in the SDK.

Now that you have a reference to the device that you want to test, you can test the device. This step is explained in Step 2.2: Test the Device.

Step 2.2: Test the Device

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

The following code fragment illustrates a way to determine device capabilities. The Enumeration application uses the following algorithm to choose a supported display mode for the given device:

1. Return a video mode description for the specified element in the **DirectDrawEnumModes** object.
2. Retrieve the bits per pixel support in the returned video mode description.
3. Retrieve the given device's rendering depth.

4. Check to see if the video mode of the specified element in the **DirectDrawEnumModes** object is compatible with the given device's rendering depth. If it is, then accept the display mode; otherwise, reject it. Skip all 8-bit modes.

The Enumeration application executes the following code to implement the algorithm:

```
For j = 1 To g_ddEnumModes.GetCount()
    ' We have not tested this display mode yet, so assume it is not compatible.
    bCompatible = False
    ' Step 1:
    g_ddEnumModes.GetItem j, g_enumInfo

    ' Step 2:
    modeDepth = g_enumInfo.ddpfPixelFormat.IRGBBitCount

    ' Step 3:
    bitDepth = deviceDesc.IDeviceRenderBitDepth

    ' Step 4:
    If (32 = modeDepth) And (bitDepth And DDBD_32) Then bCompatible = True
    If (24 = modeDepth) And (bitDepth And DDBD_24) Then bCompatible = True
    If (16 = modeDepth) And (bitDepth And DDBD_16) Then bCompatible = True

    ' Populate the combo box with only the display modes that we want.
    If bCompatible Then
        cmbMode.AddItem Str(g_enumInfo.IWidth) + " x" + Str(g_enumInfo.IHeight) + " x" +
        Str(g_enumInfo.ddpfPixelFormat.IRGBBitCount)
    End If
Next j
```

After this algorithm completes, the combo box, *cmbMode*, will hold a list display modes that meet the specified criteria, all other display modes will be ignored.

Tutorial 5: Using Texture Maps

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

This tutorial explains how to draw a simple three-dimensional (3-D) scene using textures. In essence, a Direct3D texture is a bitmap of pixel colors; as a result, any

bitmap image can be applied to a Direct3D primitive. Textures are a simple, powerful way to add realism to your 3-D objects without altering their geometry. In this way, textures can be used to enhance a primitive with patterns and illusions of color, roughness, transparency, and smoothness. Specifically, the Using Texture Maps tutorial explains how to create textures from file-based bitmaps and how to render a cube primitive with textures.

Note that texture handles are not used in Direct3D for Visual Basic; rather, you obtain a reference to the texture surface object when you create the texture surface

In order to add textures to the primitives in your 3-D scenes, complete the following steps:

- Step 1: Create a Textured Surface
- Step 2: Render a Textured Primitive

Direct3D supports an extensive texturing feature set, providing developers with easy access to advanced texturing techniques and strategies. For more information on textures and how they can be used and optimized in Direct3D, see Textures.

Note

The code in the Texture application is similar to the code in the Triangle application. This tutorial focuses only on the texturing code unique to the Texture application, and does not cover setting up Direct3D, preparing scene geometry, shutting down, or processing system events. For information on these tasks, see Tutorial 1: Rendering a Single Triangle.

The Texture tutorial draws a cube primitive. For more information on how to draw primitives, see Tutorial 3: Using Alternate Primitive Styles.

The Texture tutorial uses enumeration to enumerate the texture formats. For more information on enumeration techniques and principles, see Tutorial 4: Using Device Enumeration.

Step 1: Create a Textured Surface

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Direct3D for Visual Basic greatly simplifies the task of creating and managing textures. To prepare, create, and implement system-managed textures in your 3-D scene, follow these steps:

- Step 1.1: Set the Texture Coordinates

- Step 1.2: Prepare the Texture Surface
- Step 1.3: Enumerate the Texture Formats
- Step 1.4: Create a New Surface for the Texture

Before creating a texture for a primitive, you need to define the vertices of your primitive along with its texture coordinates. This task is defined in Step 1.1: Set the Texture Coordinates.

Step 1.1: Set the Texture Coordinates

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Direct3D represents textures as two dimensional arrays of color values. Each element in the array is referred to as a texel. You can access a texel through column and row numbers, which are labeled U and V, respectively.

Your application can assign texture coordinates directly to the vertices by using the **DirectX7.CreateD3DVertex** method. The following code fragment, taken from the Texture application, creates a set of four vertices describing the front face of a cube:

```
g_dx.CreateD3DVertex -1, 1, -1, 0, 0, -1, 0, 0, vertices(0)
g_dx.CreateD3DVertex 1, 1, -1, 0, 0, -1, 1, 0, vertices(1)
g_dx.CreateD3DVertex -1, -1, -1, 0, 0, -1, 0, 1, vertices(2)
g_dx.CreateD3DVertex 1, -1, -1, 0, 0, -1, 1, 1, vertices(3)
```

For each **CreateD3DVertex** method call, the first three arguments define a vertex, the second three arguments define the orientation of the vertex normal vector, and the next two arguments represent the texture coordinates of the vertex. In this case, you want the application to map the entire bitmap texture onto the front face of the cube, so you will use the following texture coordinates: (0, 0), (1, 0), (0, 1), and (1, 1).

For more information on texture coordinates, see Texture Coordinates.

You should employ the same procedures for the back, top, bottom, right, and left faces of the cube. See the Texture application code in the SDK for information on how to do this.

The Texture application uses the programmer-defined function `CreateTextureSurface` to return a **DirectDrawSurface7** object. The returned texture surface object will be set to the primitive during rendering:

```
Set g_ddsTexture1 = CreateTextureSurface("tree1.bmp")
Set g_ddsTexture2 = CreateTextureSurface("tex1.bmp")
Set g_ddsTexture3 = CreateTextureSurface("earth.bmp")
```

In the preceding code, three texture surface objects are created from the following bitmap files: ("tree1.bmp, " "tex1.bmp," and "earth.bmp").

Note

In order to run the Texture application, you need to ensure that your Visual Basic application can find your texture files ("tree1.bmp", "tex1.bmp", and "earth.bmp"). The Texture application looks for these files in the same folder that contains your Texture application's project file (.vbp).

Now, you can look at the programmer-defined subroutine CreateTextureSurface in detail, to see how texture surface objects are prepared and created. The initial step of this process is illustrated in Step 1.2: Prepare the Texture Surface.

Step 1.2: Prepare the Texture Surface

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

You can use a **DDSURFACEDESC2** type to describe the texture surface:

```
Public Function CreateTextureSurface(sFile As String) As DirectDrawSurface7
    Dim ddsTexture As DirectDrawSurface7
    Dim i As Long
    Dim blsFound As Boolean

    ' Prepare the texture surface.
    Dim ddsd As DDSURFACEDESC2
```

```
    ddsd.lFlags = DDSD_CAPS Or DDSD_HEIGHT Or DDSD_WIDTH Or
    DDSD_PIXELFORMAT Or DDSD_TEXTURESTAGE
```

Now, you are ready to enumerate the texture formats. This step is explained in Step 1.3: Enumerate the Texture Formats.

Step 1.3: Enumerate the Texture Formats

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Before you enumerate your textures, you need to declare a **Direct3DEnumPixelFormat** object:

```
Dim TextureEnum As Direct3DEnumPixelFormat
```

Then, you can call the **Direct3DDevice7.GetTextureFormatsEnum** method on the rendering device to return a **Direct3DEnumPixelFormat** object:

```
Set TextureEnum = g_d3dDevice.GetTextureFormatsEnum()
```

Now, you can call methods on the **Direct3DEnumPixelFormat** object, *TextureEnum*, to retrieve information about the pixel formats that the device supports for textures.

In the following code excerpt from the Texture application, the loop is looking for a device-supported texture pixel format. Specifically, the code looks for a 16-bit texture. Note that all other formats, such as alpha textures, are ignored:

```
For i = 1 To TextureEnum.GetCount()
    blsFound = True
    Call TextureEnum.GetItem(i, ddsd.ddpfPixelFormat)

    With ddsd.ddpfPixelFormat
        ' Skip unusual modes.
        If .IFlags And (DDPF_LUMINANCE Or DDPF_BUMPLUMINANCE Or
            DDPF_BUMPDUDV) Then blsFound = False

        ' Skip any FourCC formats.
        If .IFourCC <> 0 Then blsFound = False

        'Skip alpha modes.
        If .IFlags And DDPF_ALHAPIXELS Then blsFound = False

        'We only want 16-bit formats, so skip all others.
        If .IRGBBitCount <> 16 Then blsFound = False
    End With

    If blsFound Then Exit For
Next i
```

Now that a texture format supported by the rendering device has been selected from the enumeration of texture formats, *TextureEnum*, you can create a surface for the texture. This step is detailed in Step 1.4: Create a New Surface for the Texture.

Step 1.4: Create a New Surface for the Texture

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

Before creating the texture surface object, you should turn on texture management for the device with the DDSCAPS2_TEXTUREMANAGE flag:

```
ddsd.ddsCaps.ICaps = DDSCAPS_TEXTURE
ddsd.ddsCaps.ICaps2 = DDSCAPS2_TEXTUREMANAGE
ddsd.lTextureStage = 0
```

Also, by setting the control flag DDSCAPS_TEXTURE, you inform the system that the surface describes a texture surface.

For more information on texture management, see Automatic Texture Management.

Now that automatic texture management has been enabled, and you have set the DDSCAPS_TEXTURE flag, you can create a new surface for the texture and return it:

```
Set ddsTexture = g_dd.CreateSurfaceFromFile(sFile, ddsd)
```

```
' Return the newly created texture.
```

```
Set CreateTextureSurface = ddsTexture
```

For more information on texture surfaces, see Texture Surface Objects.

Since you have created a new surface for your texture, you can use that texture during rendering of the primitives. This step is done in Step 2: Render a Textured Primitive.

Step 2: Render a Textured Primitive

[C++]

This section pertains only to application development in Visual Basic. See Direct3D Immediate Mode C/C++ Tutorials.

[Visual Basic]

The only difference between rendering a textured primitive and a non-textured primitive is that you have to set the texture during the rendering of a textured primitive.

The following code excerpt from the Texture application draws the front and back faces of a cube, using the texture surface *g_ddsTexture1*:

```
g_d3dDevice.SetTexture 0, g_ddsTexture1
Call g_d3dDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, D3DFVF_VERTEX,
g_vCube(0), _
4, D3DDP_DEFAULT)
Call g_d3dDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, D3DFVF_VERTEX,
g_vCube(4), _
```

4, D3DDP_DEFAULT)

You should employ the same procedures for the top and bottom faces of the cube, and the left and right faces of the cube, setting *g_ddsTexture2* and *g_ddsTexture3*, respectively. See the Texture application code in the SDK for information on how to do this.

The **Direct3DDevice7.SetTexture** method assigns a texture to a given stage for a device. Since this tutorial does not use multiple texture blending, the first parameter of **SetTexture** is 0, specifying a value of 0 for the stage identifier. The second parameter identifies the **DirectDrawSurface7** object being set as the texture.

For more information on textured rendering, see *Rendering with Texture Surfaces*.

For more information on multiple texture blending, see *Multiple Texture Blending*.

Direct3D Immediate Mode Reference

This section contains reference information for the application programming interface (API) elements provided by Microsoft® Direct3D® Immediate Mode in C/C++ and Microsoft® Visual Basic® programming languages. Reference material is organized by language:

- Direct3D Immediate Mode C/C++ Reference
- Direct3D Immediate Mode Visual Basic Reference

Direct3D Immediate Mode C/C++ Reference

This section contains reference information for the API elements provided by Direct3D Immediate Mode. Reference material is divided into the following categories:

- Interfaces
- D3D_OVERLOADS
- Callback Functions
- Macros
- Structures
- Enumerated Types

- Other Types
- Flexible Vertex Format Flags
- Return Values

Interfaces

This section contains reference information for the COM interfaces provided by Direct3D Immediate Mode. The primary interfaces used with Direct3D Immediate Mode are the following:

- **IDirect3D7**
- **IDirect3DDevice7**
- **IDirect3DVertexBuffer7**

Note:

Direct3D no longer requires a discrete interface for textures. All the texture-related methods in Direct3D now accept pointers to the **IDirectDrawSurface7** interface, as long as the surface was created with texture capabilities (DDSCAPS_TEXTURE).

Applications that require execute buffers must use the legacy interfaces provided by previous releases of DirectX.

Some stub methods that were exposed in previous releases of DirectX are no longer exposed. For more information, see Unimplemented Methods.

The new **IDirect3DDevice7** interface includes methods that render the **IDirect3DLight**, **IDirect3DMaterial3**, and **IDirect3DViewport3** interfaces obsolete. For details, see Obsolete Interfaces. Documentation for these interfaces is available through previous releases of DirectX.

IDirect3D7

Applications use the methods of the **IDirect3D7** interface to create Direct3D objects and set up the environment. This section is a reference to the methods of this interface. For a conceptual overview, see Direct3D Interfaces.

The **IDirect3D7** interface is obtained by calling the **QueryInterface** method from a **DirectDraw** object.

The methods of the **IDirect3D7** interface can be organized into the following groups:

Creation	CreateDevice
	CreateVertexBuffer
Enumeration	EnumDevices
	EnumZBufferFormats
Miscellaneous	EvictManagedTextures

The **IDirect3D7** interface simplifies the **IDirect3D3** interface by eliminating the need for creation methods for objects other than devices and vertex buffers. In contrast to the **IDirect3D3** interface, the **IDirect3D3::CreateLight**, **IDirect3D3::CreateMaterial**, and **IDirect3D3::CreateViewport** methods are not present in **IDirect3D7**. The functionality provided by these methods is now offered by the **IDirect3DDevice7** interface.

The **IDirect3D7** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

The **LPDIRECT3D7**, **LPDIRECT3D3**, **LPDIRECT3D2**, and **LPDIRECT3D** types are defined as pointers to the **IDirect3D7**, **IDirect3D3**, **IDirect3D2**, and **IDirect3D** interfaces:

```
typedef struct IDirect3D *LPDIRECT3D;
typedef struct IDirect3D2 *LPDIRECT3D2;
typedef struct IDirect3D3 *LPDIRECT3D3;
typedef struct IDirect3D7 *LPDIRECT3D7;
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

Accessing Direct3D, Direct3D and DirectDraw

IDirect3D7::CreateDevice

The **IDirect3D7::CreateDevice** method creates a Direct3D device to be used with the DrawPrimitive methods.

```
HRESULT CreateDevice(
    REFCLSID rclsid,
    LPDIRECTDRAWSURFACE7 lpDDS,
    LPDIRECT3DDEVICE7 * lpIpd3DDevice,
);
```

Parameters

rclsid

Class identifier for the new device. This value can be IID_IDirect3DTnLHalDevice, IID_IDirect3DHALDevice, IID_IDirect3DMMXDevice, or IID_IDirect3DRGBDevice. The IID_IDirect3DRampDevice, used for the ramp emulation device, is not supported by interfaces later than **IDirect3D2**. To use ramp emulation, you must use the legacy **IDirect3D2** interface.

lpDDS

Address of the **IDirectDrawSurface7** interface for the DirectDrawSurface object that is the device's rendering target. The surface must have been created as a 3-D device by using the DDSCAPS_3DDEVICE capability.

lpD3DDevice

Address of a pointer to the new **IDirect3DDevice7** interface when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

All rendering devices created by a given 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.

In the legacy **IDirect3D3** interface, this method accepted pointers to the **IDirectDrawSurface4** and **IDirect3DDevice3** interfaces.

When you call **IDirect3D7::CreateDevice**, you create a device object that is separate from a DirectDraw surface object. This device uses a DirectDraw surface as a rendering target.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7, Creating a Direct3D Device, Direct3D Devices

IDirect3D7::CreateVertexBuffer

The **IDirect3D7::CreateVertexBuffer** method creates a vertex buffer object.

```
HRESULT CreateVertexBuffer(  
    LPD3DVERTEXBUFFERDESC lpVBDesc,  
    LPDIRECT3DVERTEXBUFFER7* lpD3DVertexBuffer,  
    DWORD dwFlags  
);
```

Parameters

lpVBDesc

Address of a **D3DVERTEXBUFFERDESC** structure that describes the format and number of vertices that the vertex buffer receives.

lpD3DVertexBuffer

Address of a variable to receive a pointer to a **IDirect3DVertexBuffer7** interface for the new vertex buffer.

dwFlags

No flags are currently defined for this method. This parameter must be 0.

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:

```
CLASS_E_NOAGGREGATION  
D3DERR_INVALIDVERTEXFORMAT  
D3DERR_VBUF_CREATE_FAILED  
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
DDERR_OUTOFMEMORY
```

Remarks

This method was introduced with the **IDirect3D3** interface. As part of **IDirect3D3**, this method accepted a pointer to the legacy **IDirect3DVertexBuffer** interface and accepted flags in the *dwFlags* parameter.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DVertexBuffer7, Vertex Buffers

IDirect3D7::EnumDevices

The **IDirect3D7::EnumDevices** method enumerates all Direct3D device drivers installed on the system.

```
HRESULT EnumDevices(  
    LPD3DENUMDEVICESCALLBACK7 lpEnumDevicesCallback,  
    LPVOID lpUserArg  
);
```

Parameters

lpEnumDevicesCallback

Address of the **D3DEnumDevicesCallback7** callback function that the enumeration procedure calls every time a match is found.

lpUserArg

Address of application-defined data passed to the callback function.

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:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

In the **IDirect3D3** interface, this method accepted a pointer to a **D3DEnumDevicesCallback** function.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3D7::EnumZBufferFormats

The **IDirect3D7::EnumZBufferFormats** method enumerates the supported depth-buffer formats for a specified device.

```
HRESULT EnumZBufferFormats(
    REFCLSID riidDevice,
    LPD3DENUMPIXELFORMATSCALLBACK lpEnumCallback,
    LPVOID lpContext
);
```

Parameters

riidDevice

Reference to a globally unique identifier for the device whose depth-buffer formats are enumerated.

lpEnumCallback

Address of a **D3DEnumPixelFormatCallback** callback function that is called for each supported depth-buffer format.

lpContext

Application-defined data that is passed to the callback function.

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:

```
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOZBUFFERHW
DDERR_OUTOFMEMORY
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3D7::EvictManagedTextures

The **IDirect3D7::EvictManagedTextures** method evicts all managed textures from local or nonlocal video memory.

HRESULT EvictManagedTextures();

Parameters

None.

Return Values

This method returns D3D_OK.

Remarks

This method causes Direct3D to remove any texture surfaces created with the DDSCAPS2_TEXTUREMANAGE or DDSCAPS2_D3DTEXTUREMANAGE flags from local or nonlocal video memory.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

Automatic Texture Management

IDirect3DDevice7

The **IDirect3DDevice7** interface provides methods enabling applications to perform DrawPrimitive-based rendering, in contrast to the **IDirect3DDevice** interface, which applications use to work with execute buffers. You can create a Direct3DDevice object and retrieve a pointer to this interface by calling the **IDirect3D7::CreateDevice** method.

For a conceptual overview, see Direct3D Devices and the DrawPrimitive Methods.

The methods of the **IDirect3DDevice7** interface can be organized into the following groups:

Information	GetCaps
	GetDirect3D
	GetInfo
Device states	ApplyStateBlock
	BeginStateBlock
	CaptureStateBlock
	CreateStateBlock
	DeleteStateBlock
	EndStateBlock
	GetClipStatus
	GetRenderState
	GetRenderTarget
	GetStateData
	GetTransform
	SetClipStatus
	SetRenderState
	SetRenderTarget
	SetStateData
	SetTransform
Lighting and materials	GetLight
	GetLightEnable
	GetMaterial
	LightEnable
	SetLight
	SetMaterial
Miscellaneous	ComputeSphereVisibility
	MultiplyTransform
Rendering	DrawIndexedPrimitive
	DrawIndexedPrimitiveStrided
	DrawIndexedPrimitiveVB
	DrawPrimitive
	DrawPrimitiveStrided
	DrawPrimitiveVB
Scenes	BeginScene

	EndScene
Textures	EnumTextureFormats
	GetTexture
	GetTextureStageState
	Load
	PreLoad
	SetTexture
	SetTextureStageState
	ValidateDevice
User-defined clip planes	GetClipPlane
	SetClipPlane
Viewports	Clear
	GetViewport
	SetViewport

The **IDirect3DDevice7** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

This interface extends the **IDirect3DDevice3** interface by including methods for device-state blocks. Device-state blocks accelerate the common sequences of state changes that your application requires. In addition, the **IDirect3DDevice7** interface includes methods to work with viewports, lights, and materials within the device that supersede the **IDirect3DViewport3**, **IDirect3DMaterial3**, and **IDirect3DLight** interfaces.

Many of the methods common to this interface and its predecessor, **IDirect3DDevice3**, accept slightly different parameters. Wherever an **IDirect3DDevice3** interface method might accept an **IDirectDrawSurface3** interface pointer as a parameter, the methods in the **IDirect3DDevice7** interface accept an **IDirectDrawSurface7** interface pointer instead. Methods that previously accepted pointers to an **IDirect3DVertexBuffer** or **IDirect3DTexture2** interface now accept **IDirect3DVertexBuffer7** or **IDirectDrawSurface7** interface pointers instead. The methods for viewports, materials, and lighting that were added to the interface accept pointers to new structures, as described in the reference page for each new method.

This interface is not intended to be used with execute buffers and therefore does not contain any execute-buffer-related methods. For information about execute buffers, see the documentation provided with a previous release of DirectX. Legacy documentation is provided with this SDK and available for download from <http://www.microsoft.com/directx>.

You can use the **LPDIRECT3DDEVICE2**, **LPDIRECT3DDEVICE3**, or **LPDIRECT3DDEVICE7** data types to declare a variable that contains a pointer to an **IDirect3DDevice2**, **IDirect3DDevice3**, or **IDirect3DDevice7** interface. The **D3d.h** header file declares these data types with the following code:

```
typedef struct IDirect3DDevice2 *LPDIRECT3DDEVICE2;
typedef struct IDirect3DDevice3 *LPDIRECT3DDEVICE3;
typedef struct IDirect3DDevice7 *LPDIRECT3DDEVICE7;
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in **d3d.h**.

See Also

Direct3D Devices, Rendering

IDirect3DDevice7::ApplyStateBlock

The **IDirect3DDevice7::ApplyStateBlock** method applies an existing device-state block to the rendering device.

```
HRESULT ApplyStateBlock(
    DWORD dwBlockHandle
);
```

Parameters

dwBlockHandle

Handle to the device state block to be executed, as returned by a previous call to the **IDirect3DDevice7::EndStateBlock** method.

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_INBEGINSTATEBLOCK
D3DERR_INVALIDSTATEBLOCK
```

Remarks

Applications cannot apply a device state block while recording another block.

This method was introduced with the **IDirect3DDevice7** interface.

As with all operations that affect the state of the rendering device, it is recommend that you apply state blocks during scene rendering—that is, after calling the **IDirect3DDevice7::BeginScene** method and before calling **IDirect3DDevice7::EndScene**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::BeginStateBlock, **IDirect3DDevice7::EndStateBlock**, **IDirect3DDevice7::CaptureStateBlock**, **IDirect3DDevice7::CreateStateBlock**, **IDirect3DDevice7::DeleteStateBlock**, State Blocks

IDirect3DDevice7::BeginScene

The **IDirect3DDevice7::BeginScene** method begins a scene. Applications must call this method before performing any rendering and must call **IDirect3DDevice7::EndScene** when rendering is complete and before calling **IDirect3DDevice7::BeginScene** again.

HRESULT BeginScene();

Parameters

None.

Return Values

If the method succeeds, the return value is **D3D_OK**.

If the method fails, the return value is an error. For a complete list of Direct3D Immediate Mode error values and descriptions, see Return Values.

Remarks

If the **BeginScene** method fails, the device was unable to begin the scene, and there is no need to call the **IDirect3DDevice7::EndScene** method. In fact, calls to the **IDirect3DDevice7::EndScene** method will fail if the previous call to **BeginScene** failed.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::EndScene, Beginning and Ending a Scene

IDirect3DDevice7::BeginStateBlock

The **IDirect3DDevice7::BeginStateBlock** method signals 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_INBEGINSTATEBLOCK  
DDERR_OUTOFMEMORY
```

Remarks

This method was introduced with the **IDirect3DDevice7** interface.

Applications can ensure that all recorded states are valid by calling the **IDirect3DDevice7::ValidateDevice** method prior to calling this method.

The following methods can be recorded in a state block (that is, after calling **IDirect3DDevice7::BeginStateBlock** and before **IDirect3DDevice7::EndStateBlock**):

- **IDirect3DDevice7::LightEnable**
- **IDirect3DDevice7::SetClipPlane**
- **IDirect3DDevice7::SetLight**
- **IDirect3DDevice7::SetMaterial**
- **IDirect3DDevice7::SetRenderState**
- **IDirect3DDevice7::SetTexture**
- **IDirect3DDevice7::SetTextureStageState**
- **IDirect3DDevice7::SetTransform**
- **IDirect3DDevice7::SetViewport**

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::ApplyStateBlock, **IDirect3DDevice7::EndStateBlock**, **IDirect3DDevice7::CaptureStateBlock**, **IDirect3DDevice7::CreateStateBlock**, **IDirect3DDevice7::DeleteStateBlock**, State Blocks

IDirect3DDevice7::CaptureStateBlock

The **IDirect3DDevice7::CaptureStateBlock** method updates the values within an existing state block to the values currently set for the device.

```
HRESULT CaptureStateBlock(  
    DWORD dwBlockHandle  
);
```

Parameters

dwBlockHandle

Handle to the state block into which the current device state is captured.

Return Values

If the method succeeds, the return value is `D3D_OK`.

If the method fails, the return value can be a standard error code, or `D3DERR_INBEGINSTATEBLOCK` if a state block is currently being recorded.

Remarks

This method captures updated values for states within an existing state block. It does not capture the entire state of the device. For more information, see [Capturing State Blocks](#).

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in `d3d.h`.

See Also

`IDirect3DDevice7::ApplyStateBlock`, `IDirect3DDevice7::BeginStateBlock`, `IDirect3DDevice7::CreateStateBlock`, `IDirect3DDevice7::EndStateBlock`, `IDirect3DDevice7::DeleteStateBlock`, [State Blocks](#), [Capturing State Blocks](#)

IDirect3DDevice7::CreateStateBlock

The `IDirect3DDevice7::CreateStateBlock` method creates a new state block that contains the current values for all device states, vertex-related states, or pixel-related states.

```
HRESULT CreateStateBlock(  
    D3DSTATEBLOCKTYPE d3dsbType,  
    LPDWORD lpdwBlockHandle  
);
```

Parameters

d3dsbType

Type of state data that the method should capture. This parameter can be set to one of the values defined in the `D3DSTATEBLOCKTYPE` enumerated type.

lpdwBlockHandle

Address of a variable 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_INBEGINSTATEBLOCK`
`DDERR_INVALIDPARAMS`
`DDERR_OUTOFMEMORY`

Remarks

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. For information about the states in each group, see [Creating Predefined State Blocks](#).

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in `d3d.h`.

See Also

`IDirect3DDevice7::ApplyStateBlock`, `IDirect3DDevice7::BeginStateBlock`, `IDirect3DDevice7::CaptureStateBlock`, `IDirect3DDevice7::EndStateBlock`, `IDirect3DDevice7::DeleteStateBlock`, [State Blocks](#)

IDirect3DDevice7::Clear

The `IDirect3DDevice7::Clear` method 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. This method supersedes the clearing methods exposed by the legacy `IDirect3DViewport3` interface.

```
HRESULT Clear(
    DWORD    dwCount,
    LPD3DRECT lpRects,
    DWORD    dwFlags,
    DWORD    dwColor,
    D3DVALUE dvZ,
    DWORD    dwStencil
```

);

Parameters

dwCount

Number of rectangles in the array at *lpRects*. If you set *lpRects* to NULL, this parameter must be set to 0.

lpRects

Array of **D3DRECT** structures that describe the rectangles to be cleared. 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.

dwFlags

Flags indicating which surfaces should be cleared. This parameter can be any combination of the following flags, but at least one flag must be used:

D3DCLEAR_TARGET

Clear the rendering target to the color in the *dwColor* parameter.

D3DCLEAR_STENCIL

Clear the stencil buffer to the value in the *dwStencil* parameter.

D3DCLEAR_ZBUFFER

Clear the depth buffer to the value in the *dvZ* parameter.

dwColor

A 32-bit RGBA color value to which the render target surface is cleared.

dvZ

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.

dwStencil

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 one of the following values:

D3DERR_STENCILBUFFER_NOTPRESENT

D3DERR_VIEWPORTHASNODEVICE

D3DERR_ZBUFFER_NOTPRESENT

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

This method was introduced with the **IDirect3DDevice7** interface.

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3DDevice7::ComputeSphereVisibility

The **IDirect3DDevice7::ComputeSphereVisibility** method calculates the visibility (complete, partial, or no visibility) of an array of spheres within the current viewport for this device.

```
HRESULT ComputeSphereVisibility(
    LPD3DVECTOR lpCenters,
    LPD3DVALUE lpRadii,
    DWORD dwNumSpheres,
    DWORD dwFlags,
    LPDWORD lpdwReturnValues
);
```

Parameters

lpCenters

Array of **D3DVECTOR** structures describing the center point for each sphere, in world-space coordinates.

lpRadii

Array of **D3DVALUE** variables that represent the radius for each sphere.

dwNumSpheres

Number of spheres. This value must be greater than 0.

dwFlags

Not currently used; set to 0.

lpdwReturnValues

Array of **DWORD** values. The array need not be initialized, but it must be large enough to contain a **DWORD** for each sphere being tested. When the method returns, each element in the array contains a combination of flags that describe the visibility of that sphere within the current viewport for this device. If a sphere is completely visible, the corresponding entry in *lpdwReturnValues* is 0. The following flags can be combined and present in the array:

Basic Clipping Flags

D3DCLIP_BACK

All vertices are clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM

All vertices are clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT

All vertices are clipped by the front plane of the viewing frustum.

D3DCLIP_LEFT

All vertices are clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT

All vertices are clipped by the right plane of the viewing frustum.

D3DCLIP_TOP

All vertices are clipped by the top plane of the viewing frustum.

D3DCLIP_GEN0 through D3DCLIP_GEN5

All vertices are clipped by a corresponding application-defined clipping plane.

Combination and General Flags

D3DSTATUS_CLIPINTERSECTIONALL

Combination of all CLIPINTERSECTION flags.

D3DSTATUS_CLIPUNIONALL

Combination of all CLIPUNION flags.

D3DSTATUS_DEFAULT

Combination of D3DSTATUS_CLIPINTERSECTIONALL and D3DSTATUS_ZNOTVISIBLE flags. This value is the default.

D3DSTATUS_ZNOTVISIBLE

Indicates that the rendered primitive is not visible. This flag is set or cleared by the system when rendering with z-checking enabled (see D3DRENDERSTATE_ZVISIBLE).

Clip Intersection Flags

D3DSTATUS_CLIPINTERSECTIONBACK

Logical **And** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM

Logical **And** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT

Logical **And** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through

D3DSTATUS_CLIPINTERSECTIONGEN5

Logical **And** of the clip flags for application-defined clipping planes.

D3DSTATUS_CLIPINTERSECTIONLEFT

Logical **And** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT

Logical **And** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP

Logical **And** of the clip flags for the vertices compared to the top of the viewing frustum.

Clip Union Flags

D3DSTATUS_CLIPUNIONBACK

Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM

Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT

Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5

Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT

Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT

Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP

Equal to D3DCLIP_TOP.

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_INVALIDMATRIX

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

Sphere visibility is computed by back-transforming the viewing frustum to the model space, using the inverse of the combined world, view, or projection matrices. If the combined matrix cannot be inverted (if the determinant is 0), the method fails, returning D3DERR_INVALIDMATRIX.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3DDevice7::DeleteStateBlock

The **IDirect3DDevice7::DeleteStateBlock** method deletes a previously recorded device state block.

```
HRESULT DeleteStateBlock (  
    DWORD dwBlockHandle  
);
```

Parameters

dwBlockHandle

Handle to the device state block to be deleted, as returned by a previous call to the **IDirect3DDevice7::EndStateBlock** method.

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_INBEGINSTATEBLOCK

D3DERR_INVALIDSTATEBLOCK

Remarks

Applications cannot delete a device state block while another is being recorded.

This method was introduced with the **IDirect3DDevice7** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::ApplyStateBlock, **IDirect3DDevice7::BeginStateBlock**,
IDirect3DDevice7::CaptureStateBlock, **IDirect3DDevice7::CreateStateBlock**,
IDirect3DDevice7::EndStateBlock, State Blocks

IDirect3DDevice7::DrawIndexedPrimitive

The **IDirect3DDevice7::DrawIndexedPrimitive** method renders the specified geometric primitive, based on indexing into an array of vertices.

```
HRESULT DrawIndexedPrimitive(
    D3DPRIMITIVETYPE d3dptPrimitiveType,
    DWORD dwVertexTypeDesc,
    LPVOID lpvVertices,
    DWORD dwVertexCount,
    LPWORD lpwIndices,
    DWORD dwIndexCount,
    DWORD dwFlags
);
```

Parameters

d3dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

The **D3DPT_POINTLIST** member of **D3DPRIMITIVETYPE** is not indexed.

dwVertexTypeDesc

A combination of flexible vertex format flags that describes the vertex format for this set of primitives.

lpvVertices

Pointer to the list of vertices to be used in the primitive sequence.

dwVertexCount

Defines the number of vertices in the list.

This parameter is used differently from the *dwVertexCount* parameter in the **IDirect3DDevice7::DrawPrimitive** method. In that method, the *dwVertexCount* parameter gives the number of vertices to draw, but here it gives the total number of vertices in the array pointed to by the *lpvVertices* parameter. When you call **IDirect3DDevice7::DrawIndexedPrimitive**, you specify the number of vertices to draw in the *dwIndexCount* parameter.

lpwIndices

Pointer to a list of **WORD**s that are to be used to index into the specified vertex list when creating the geometry to render.

dwIndexCount

Specifies the number of indices provided for creating the geometry. The maximum number of indices allowed is D3DMAXNUMVERTICES (0xFFFF).

dwFlags

Zero to render the primitive without waiting, or the following flag:

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up to date before continuing.

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_INVALIDDRAMTEXTURE

D3DERR_INVALIDPRIMITIVETYPE

D3DERR_INVALIDVERTEXTYPE

DDERR_INVALIDPARAMS

DDERR_WASSTILLDRAWING

Remarks

Make sure that the vertices being rendered match the vertex format that you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

This method, unlike its predecessor in previous interfaces, does not accept the D3DDP_DONOTCLIP, D3DDP_DONOTLIGHT, and D3DDP_DONOTUPDATEEXTENTS flags in the *dwFlags* parameter. The functionality offered by these flags is now available through the **D3DRENDERSTATE_CLIPPING**, **D3DRENDERSTATE_LIGHTING**, and **D3DRENDERSTATE_EXTENTS** render states.

Do not use this method to render very small subsets of vertices from extremely large vertex arrays. This method transforms every vertex in the provided buffer, regardless of the location or number of vertices being rendered. Thus, if you pass an array that contains thousands of vertices, but only intend to render hundreds, your application's performance suffers dramatically. If you need to render a small number of vertices from a large buffer, use the Direct3D vertex buffer rendering methods. For more information, see Vertex Buffers.

This method differs from its counterpart in the **IDirect3DDevice2** interface in that it accepts a flexible vertex format descriptor, rather than a member of the **D3DVERTEXTYPE** enumerated type as the second parameter. If you attempt to use

one of the members of **D3DVERTEXTYPE**, the method fails, returning **DDERR_INVALIDPARAMS**. For more information, see Vertex Formats.

In current versions of DirectX, **IDirect3DDevice7::DrawIndexedPrimitive** can sometimes generate an update rectangle that is larger than it needs to be. If a large number of vertices need to be processed, this can have a negative impact on the performance of your application. If you are using **D3DTLVERTEX** vertices and the system is processing more vertices than you need, use the **D3DDP_DONOTCLIP** and **D3DDP_DONOTUPDATEEXTENTS** flags to solve the problem.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::DrawPrimitive, **IDirect3DDevice7::DrawPrimitiveStrided**, **IDirect3DDevice7::DrawPrimitiveVB**, **IDirect3DDevice7::DrawIndexedPrimitiveStrided**, **IDirect3DDevice7::DrawIndexedPrimitiveVB**

IDirect3DDevice7::DrawIndexedPrimitiveStrided

The **IDirect3DDevice7::DrawIndexedPrimitiveStrided** method renders a geometric primitive, based on indexing into an array of strided vertices. For more information, see Strided Vertex Format.

```
HRESULT DrawIndexedPrimitiveStrided(
    D3DPRIMITIVETYPE d3dptPrimitiveType,
    DWORD dwVertexTypeDesc,
    LPD3DDRAWPRIMITIVESTRIDEDDATA lpVertexArray,
    DWORD dwVertexCount,
    LPWORD lpwIndices,
    DWORD dwIndexCount,
    DWORD dwFlags
);
```

Parameters

d3dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

The D3DPT_POINTLIST member of **D3DPRIMITIVETYPE** is not indexed.

dwVertexTypeDesc

A combination of flexible vertex format flags that describes the vertex format for this primitive.

lpVertexArray

Address of a **D3DDRAWPRIMITIVESTRIDEDDATA** structure that contains pointers and memory strides for the vertex components of this primitive, in the format specified by the flags in *dwVertexTypeDesc*.

dwVertexCount

Defines the number of vertices in the list.

This parameter is used differently from the *dwVertexCount* parameter in the **IDirect3DDevice7::DrawPrimitive** method. In that method, the *dwVertexCount* parameter gives the number of vertices to draw, but here it gives the total number of vertices in the array pointed to by the *lpVertexArray* parameter. When you call **IDirect3DDevice7::DrawIndexedPrimitiveStrided**, you specify the number of vertices to draw in the *dwIndexCount* parameter.

lpwIndices

Pointer to a list of **WORDS** that are to be used to index into the specified vertex list when creating the geometry to render.

dwIndexCount

Specifies the number of indices provided for creating the geometry. The maximum number of indices allowed is D3DMAXNUMVERTICES (0xFFFF).

dwFlags

Zero to render the primitive without waiting, or the following flag:

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up to date before continuing.

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_INVALIDDRAMPTEXTURE

D3DERR_INVALIDPRIMITIVETYPE

D3DERR_INVALIDVERTEXTYPE

DDERR_INVALIDPARAMS

DDERR_WASSTILLDRAWING

Remarks

Be sure that the vertices being rendered match the vertex format that you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

This method, unlike its predecessor in previous interfaces, does not accept the `D3DDP_DONOTCLIP`, `D3DDP_DONOTLIGHT`, and `D3DDP_DONOTUPDATEEXTENTS` flags in the *dwFlags* parameter. The functionality offered by these flags is now available through the `D3DRENDERSTATE_CLIPPING`, `D3DRENDERSTATE_LIGHTING`, and `D3DRENDERSTATE_EXTENTS` render states.

This method does not support transformed vertices. Therefore, if you include the `D3DFVF_XYZRHW` vertex format descriptor in the *dwVertexTypeDesc* parameter, the method fails, returning `D3DERR_INVALIDTEXTYPE`.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in `d3d.h`.

See Also

`IDirect3DDevice7::DrawPrimitive`, `IDirect3DDevice7::DrawPrimitiveStrided`, `IDirect3DDevice7::DrawPrimitiveVB`, `IDirect3DDevice7::DrawIndexedPrimitive`, `IDirect3DDevice7::DrawIndexedPrimitiveVB`, Strided Vertex Format

IDirect3DDevice7::DrawIndexedPrimitiveVB

The `IDirect3DDevice7::DrawIndexedPrimitiveVB` method renders a geometric primitive based on indexing into an array of vertices within a vertex buffer.

```
HRESULT DrawIndexedPrimitiveVB(
    D3DPRIMITIVETYPE    d3dptPrimitiveType,
    LPDIRECT3DVERTEXBUFFER7 lpd3dVertexBuffer,
    DWORD dwStartVertex,
    DWORD dwNumVertices,
    LPWORD lpwIndices,
    DWORD dwIndexCount,
    DWORD dwFlags)
```

);

Parameters

d3dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

The D3DPT_POINTLIST member of **D3DPRIMITIVETYPE** is not indexed.

lpd3dVertexBuffer

Address of the **IDirect3DVertexBuffer7** interface for the vertex buffer that contains the array of vertices. Vertices can be transformed or untransformed, optimized or unoptimized.

dwStartVertex

Index of the first vertex in the vertex buffer to be rendered.

dwNumVertices

Total number of vertices in the vertex buffer to be rendered.

lpwIndices

Address of an array of **WORDs** that is used to index into the vertices in the vertex buffer. The values in the array must index vertices within the range [0, *dwNumVertices* - 1].

dwIndexCount

The number of indices in the array at *lpwIndices*. The maximum number of indices allowed is D3DMAXNUMVERTICES (0xFFFF).

dwFlags

Zero to render the primitive without waiting, or the following flag:

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up to date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value can be one of the following values:

D3DERR_INVALIDPRIMITIVETYPE

D3DERR_INVALIDTEXTTYPE

D3DERR_VERTEXBUFFERLOCKED

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_WASSTILLDRAWING

Remarks

Be sure that the vertices being rendered match the vertex format that you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

This method, unlike its predecessor in previous interfaces, does not accept the **D3DDP_DONOTCLIP**, **D3DDP_DONOTLIGHT**, and **D3DDP_DONOTUPDATEEXTENTS** flags in the *dwFlags* parameter. The functionality offered by these flags is now available through the **D3DRENDERSTATE_CLIPPING**, **D3DRENDERSTATE_LIGHTING**, and **D3DRENDERSTATE_EXTENTS** render states.

Software devices—MMX and RGB devices—cannot render from a video memory (local or nonlocal) vertex buffer. To render a vertex buffer, using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

You cannot render from a locked vertex buffer; calls to the **IDirect3DDevice7::DrawIndexedPrimitiveVB** or **IDirect3DDevice7::DrawPrimitiveVB** method, using a locked buffer fail, returning **D3DERR_VERTEXBUFFERLOCKED**.

In the **IDirect3DDevice3** interface, this method accepted a pointer to the legacy **IDirect3DVertexBuffer** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::DrawPrimitive, **IDirect3DDevice7::DrawPrimitiveStrided**, **IDirect3DDevice7::DrawPrimitiveVB**, **IDirect3DDevice7::DrawIndexedPrimitive**, **IDirect3DDevice7::DrawIndexedPrimitiveStrided**

IDirect3DDevice7::DrawPrimitive

The **IDirect3DDevice7::DrawPrimitive** method renders the specified array of vertices as a sequence of geometric primitives of the specified type.

HRESULT DrawPrimitive(

```

D3DPRIMITIVE dptPrimitiveType,
DWORD dwVertexTypeDesc,
LPVOID lpVertices,
DWORD dwVertexCount,
DWORD dwFlags
);

```

Parameters

dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVE** enumerated type.

dwVertexTypeDesc

A combination of flexible vertex format flags that describes the vertex format used for this set of primitives.

lpVertices

Pointer to the array of vertices to be used in the primitive sequence.

dwVertexCount

The number of vertices in the array. The maximum number of vertices allowed is D3DMAXNUMVERTICES (0xFFFF).

dwFlags

Zero to render the primitive without waiting, or the following flag:

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up to date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value can be one of the following values:

```

D3DERR_INVALIDDRAMPTEXTURE
D3DERR_INVALIDPRIMITIVETYPE
D3DERR_INVALIDVERTEXTYPE
DDERR_INVALIDPARAMS
DDERR_WASSTILLDRAWING

```

Remarks

Be sure that the vertices being rendered match the vertex format that you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride

match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

This method, unlike its predecessor in previous interfaces, does not accept the **D3DDP_DONOTCLIP**, **D3DDP_DONOTLIGHT**, and **D3DDP_DONOTUPDATEEXTENTS** flags in the *dwFlags* parameter. The functionality offered by these flags is now available through the **D3DRENDERSTATE_CLIPPING**, **D3DRENDERSTATE_LIGHTING**, and **D3DRENDERSTATE_EXTENTS** render states.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Version: Requires DirectX 7.0.
Header: Declared in d3d.h.

See Also

IDirect3DDevice7::DrawPrimitiveStrided,
IDirect3DDevice7::DrawPrimitiveVB,
IDirect3DDevice7::DrawIndexedPrimitive,
IDirect3DDevice7::DrawIndexedPrimitiveStrided,
IDirect3DDevice7::DrawIndexedPrimitiveVB

IDirect3DDevice7::DrawPrimitiveStrided

The **IDirect3DDevice7::DrawPrimitiveStrided** method renders the specified array of strided vertices as a sequence of geometric primitives. For more information, see Strided Vertex Format.

```
HRESULT DrawPrimitiveStrided(  
    D3DPRIMITIVETYPE dptPrimitiveType,  
    DWORD dwVertexTypeDesc,  
    LPD3DDRAWPRIMITIVESTRIDEDDATA lpVertexArray,  
    DWORD dwVertexCount,  
    DWORD dwFlags  
);
```

Parameters

dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

dwVertexTypeDesc

A combination of flexible vertex format flags that describes the vertex format.

lpVertexArray

Address of a **D3DDRAWPRIMITIVESTRIDEDDATA** structure that contains pointers and memory strides for the vertex components for this primitive, in the format specified by the flags in *dwVertexTypeDesc*.

dwVertexCount

Number of vertices in the array at *lpVertexArray*. The maximum number of vertices allowed is D3DMAXNUMVERTICES (0xFFFF).

dwFlags

Zero to render the primitive without waiting, or the following flag:

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up to date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value can be one of the following values:

D3DERR_INVALIDDRAMTEXTURE

D3DERR_INVALIDPRIMITIVETYPE

D3DERR_INVALIDVERTEXTYPE

DDERR_INVALIDPARAMS

DDERR_WASSTILLDRAWING

Remarks

Be sure that the vertices being rendered match the vertex format that you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

This method, unlike its predecessor in previous interfaces, does not accept the **D3DDP_DONOTCLIP**, **D3DDP_DONOTLIGHT**, and **D3DDP_DONOTUPDATEEXTENTS** flags in the *dwFlags* parameter. The functionality offered by these flags is now available through the **D3DRENDERSTATE_CLIPPING**, **D3DRENDERSTATE_LIGHTING**, and **D3DRENDERSTATE_EXTENTS** render states.

This method does not support transformed vertices. Therefore, if you include the **D3DFVF_XYZRHW** vertex format descriptor in the *dwVertexTypeDesc* parameter, the method fails, returning **D3DERR_INVALIDVERTEXTYPE**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::DrawPrimitive, **IDirect3DDevice7::DrawPrimitiveVB**,
IDirect3DDevice7::DrawIndexedPrimitive,
IDirect3DDevice7::DrawIndexedPrimitiveStrided,
IDirect3DDevice7::DrawIndexedPrimitiveVB, Strided Vertex Format

IDirect3DDevice7::DrawPrimitiveVB

The **IDirect3DDevice7::DrawPrimitiveVB** method renders an array of vertices in a vertex buffer as a sequence of geometric primitives.

```
HRESULT DrawPrimitiveVB(  
    D3DPRIMITIVETYPE    d3dptPrimitiveType,  
    LPDIRECT3DVERTEXBUFFER7 lpd3dVertexBuffer,  
    DWORD dwStartVertex,  
    DWORD dwNumVertices,  
    DWORD dwFlags  
);
```

Parameters

d3dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

lpd3dVertexBuffer

Address of the **IDirect3DVertexBuffer7** interface for the vertex buffer that contains the array of vertices. Vertices can be transformed or untransformed, optimized or unoptimized.

dwStartVertex

Index value of the first vertex in the primitive. The highest possible starting index is **D3DMAXNUMVERTICES** (0xFFFF). In debug builds, specifying a starting index value that exceeds this limit causes the method to fail and return **DDERR_INVALIDPARAMS**.

dwNumVertices

Number of vertices to be rendered. The maximum number of vertices allowed is **D3DMAXNUMVERTICES** (0xFFFF).

dwFlags

Zero to render the primitive without waiting, or the following flag:

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up to date before continuing.

Return Values

If the method succeeds, the return value is **DD_OK**.

If the method fails, the return value can be one of the following values:

D3DERR_INVALIDDRAMPTEXTURE

D3DERR_INVALIDPRIMITIVETYPE

D3DERR_INVALIDVERTEXTYPE

D3DERR_VERTEXBUFFERLOCKED

DDERR_INVALIDPARAMS

DDERR_WASSTILLDRAWING

Remarks

Be sure that the vertices being rendered match the vertex format that you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

This method, unlike its predecessor in previous interfaces, does not accept the **D3DDP_DONOTCLIP**, **D3DDP_DONOTLIGHT**, and **D3DDP_DONOTUPDATEEXTENTS** flags in the *dwFlags* parameter. The functionality offered by these flags is now available through the **D3DRENDERSTATE_CLIPPING**, **D3DRENDERSTATE_LIGHTING**, and **D3DRENDERSTATE_EXTENTS** render states.

Software devices—MMX and RGB devices—cannot render from a video memory (local or nonlocal) vertex buffer. To render a vertex buffer, using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

You cannot render from a locked vertex buffer; calls to the **IDirect3DDevice7::DrawIndexedPrimitiveVB** or **IDirect3DDevice7::DrawPrimitiveVB** method, using a locked buffer fail, returning **D3DERR_VERTEXBUFFERLOCKED**.

In the **IDirect3DDevice3** interface, this method accepted a pointer to the legacy **IDirect3DVertexBuffer** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::DrawPrimitive, **IDirect3DDevice7::DrawPrimitiveStrided**, **IDirect3DDevice7::DrawIndexedPrimitive**, **IDirect3DDevice7::DrawIndexedPrimitiveStrided**, **IDirect3DDevice7::DrawIndexedPrimitiveVB**

IDirect3DDevice7::EndScene

The **IDirect3DDevice7::EndScene** method ends a scene that was begun by calling the **IDirect3DDevice7::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 is an error.

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 **IDirect3DDevice7::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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::BeginScene, Beginning and Ending a Scene

IDirect3DDevice7::EndStateBlock

The **IDirect3DDevice7::EndStateBlock** method signals Direct3D to stop recording a device state block and retrieve a handle to the state block.

```
HRESULT EndStateBlock(  
    LPDWORD lpdwBlockHandle  
);
```

Parameters

lpdwBlockHandle

Address of a variable to be filled with the handle to the completed device state block. This value is used with the **IDirect3DDevice7::ApplyStateBlock** and **IDirect3DDevice7::DeleteStateBlock** methods.

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_NOTINBEGINSTATEBLOCK
DDERR_INVALIDPARAMS

Remarks

This method was introduced with the **IDirect3DDevice7** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::ApplyStateBlock, **IDirect3DDevice7::BeginStateBlock**,
IDirect3DDevice7::CaptureStateBlock, **IDirect3DDevice7::CreateStateBlock**,
IDirect3DDevice7::DeleteStateBlock, State Blocks

IDirect3DDevice7::EnumTextureFormats

The **IDirect3DDevice7::EnumTextureFormats** method queries the current driver for a list of supported texture formats.

```
HRESULT EnumTextureFormats(
    LPD3DENUMPIXELFORMATSCALLBACK lpd3dEnumPixelProc,
    LPVOID lpArg
);
```

Parameters

lpd3dEnumPixelProc

Address of the **D3DEnumPixelFormatFormatsCallback** callback function that the enumeration procedure calls for each texture format.

lpArg

Address of application-defined data passed to the callback function.

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:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

In the **IDirect3DDevice2** interface, this method accepts a pointer to the legacy **D3DEnumTextureFormatsCallback** function, not a **D3DEnumPixelFormatFormatsCallback**.

This method might not enumerate newly implemented texture formats on some devices. Applications that require a texture format that is not enumerated can attempt to create a surface of that format. If the creation attempt succeeds, the format is supported.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3DDevice7::GetCaps

The **IDirect3DDevice7::GetCaps** method retrieves the capabilities of the Direct3D device.

```
HRESULT GetCaps(  
    LPD3DDEVICEDESC7 lpD3DDevDesc,  
);
```

Parameters

lpD3DDevDesc

Address of the **D3DDEVICEDESC7** structure to contain the hardware features 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:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

This method does not retrieve the capabilities of the display device. To retrieve this information, use the **IDirectDraw7::GetCaps** method.

In previous versions of this interface, this method simultaneously retrieved capabilities for hardware abstraction layer (HAL) devices and hardware emulation layer (HEL) devices by accepting pointers to the legacy **D3DDEVICEDESC** structure. In the **IDirect3DDevice7** interface, this method only retrieves the capabilities of the device on which the method is called.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3DDevice7::GetClipPlane

The **IDirect3DDevice7::GetClipPlane** method retrieves the coefficients of a user-defined clipping plane for the device.

```
HRESULT GetClipPlane(  
    DWORD dwIndex,  
    D3DVALUE* pPlaneEquation  
);
```

Parameters

dwIndex

Index of the clipping plane for which the plane equation coefficients are retrieved.

pPlaneEquation

Address of 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 **DDERR_INVALIDPARAMS**. This error indicates that the value in *dwIndex* exceeds the maximum clipping plane index supported by the device or that the array at *pPlaneEquation* 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 *pPlaneEquation* 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 $Ax + By + Cz + D = 0$. A point with homogeneous coordinates (x, y, z, w) is visible in the half space of the plane if $Ax + By + Cz + Dw \geq 0$. 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 **IDirect3DDevice7::SetClipPlane** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::SetClipPlane, D3DRENDERSTATE_CLIPPLANEENABLE, User-defined Clip Planes

IDirect3DDevice7::GetClipStatus

The **IDirect3DDevice7::GetClipStatus** method gets the current clip status.

```
HRESULT GetClipStatus(  
    LPD3DCLIPSTATUS lpD3DClipStatus  
);
```

Parameters

lpD3DClipStatus

Address of a **D3DCLIPSTATUS** structure that describes the current clip status.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::SetClipStatus

IDirect3DDevice7::GetDirect3D

The **IDirect3DDevice7::GetDirect3D** method retrieves the Direct3D object for this device.

```
HRESULT GetDirect3D(  
    LPDIRECT3D7 *lpD3D  
);
```

Parameters

lpD3D

Address that receives a pointer to the Direct3D object's **IDirect3D7** interface when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return values, see Direct3D Immediate Mode Return Values.

Remarks

In previous versions of this interface, this method accepted a pointer to the legacy **IDirect3D3** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3DDevice7::GetInfo

The **IDirect3DDevice7::GetInfo** method retrieves information about the rendering device. Information can pertain to Direct3D or the underlying device driver.

```
HRESULT GetInfo(  
    DWORD dwDevInfoID,  
    LPVOID pDevInfoStruct,  
    DWORD dwSize  
);
```

Parameters

dwDevInfoID

Flags indicating the type of device information structure at *pDevInfoStruct*. This parameter can be set to one of the following flags:

D3DDEVINFOID_TEXTUREMANAGER

The structure at *pDevInfoStruct* is a

D3DDEVINFO_TEXTUREMANAGER structure that contains information about the texture management being performed by the driver. If the driver

does not manage textures, information about texture management performed by Direct3D is retrieved.

D3DDEVINFOID_D3DTEXTUREMANAGER

The structure at *pDevInfoStruct* is a **D3DDEVINFO_TEXTUREMANAGER** structure that contains information about texture management that is performed by Direct3D.

D3DDEVINFOID_TEXTURING

The structure at *pDevInfoStruct* is a **D3DDEVINFO_TEXTURING** structure that contains information about texturing activity of the application.

pDevInfoStruct

Address of a structure that receives the specified device information if the call succeeds. The type of structure, and how the data it contains is to be interpreted, is determined by the flag in *dwDevInfoID*.

dwSize

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 DirectX.

If the method fails, the return value can be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

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 your application's performance when it executes slowly. Do not call this method during scene rendering (between calls to **IDirect3DDevice7::BeginScene** and **IDirect3DDevice7::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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

D3DDEVINFO_TEXTUREMANAGER

IDirect3DDevice7::GetLight

The **IDirect3DDevice7::GetLight** method retrieves a set of lighting properties that this device uses.

```
HRESULT GetLight(  
    DWORD dwLightIndex,  
    LPD3DLIGHT7 lpLight  
);
```

Parameters

dwLightIndex

Zero-based index of the lighting property set to be retrieved.

lpLight

Address of a **D3DLIGHT7** structure that is filled with the retrieved lighting-parameter set.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if the *lpLight* parameter is invalid.

Remarks

This method was introduced with the **IDirect3DDevice7** interface.

Unlike its predecessors, the **IDirect3DDevice7** interface does not use light objects. This method, and its use of the **D3DLIGHT7** structure to describe a set of lighting properties, replaces the lighting semantics used by previous versions of the device interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::SetLight, **IDirect3DDevice7::GetLightEnable**,
IDirect3DDevice7::LightEnable

IDirect3DDevice7::GetLightEnable

The **IDirect3DDevice7::GetLightEnable** method retrieves the activity status—enabled or disabled—for a set of lighting parameters within a device.

```
HRESULT GetLightEnable(  
    DWORD dwLightIndex,  
    BOOL* pbEnable  
);
```

Parameters

dwLightIndex

Zero-based index of the set of lighting parameters that are the target of this method.

pbEnable

Address of a variable to be filled 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 is an error. For a complete list of Direct3D Immediate Mode error values and descriptions, see Return Values.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetLight, **IDirect3DDevice7::LightEnable**,
IDirect3DDevice7::SetLight

IDirect3DDevice7::GetMaterial

The **IDirect3DDevice7::GetMaterial** method retrieves the current material properties for the device.

```
HRESULT GetMaterial(  
    LPD3DMATERIAL7 lpMaterial  
);
```

Parameters

lpMaterial

Address of a **D3DMATERIAL7** structure to be filled with the currently set material properties.

Return Values

If the method succeeds, the return value is **D3D_OK**.

If the method fails, the return value is an error. The method returns **DDERR_INVALIDPARAMS** if the *lpMaterial* parameter is invalid.

Remarks

This method was introduced with the **IDirect3DDevice7** interface.

Unlike its predecessors, the **IDirect3DDevice7** interface does not use material handles. This method, and its use of the **D3DMATERIAL7** structure to describe material properties, replaces the material-handle semantics used by previous versions of the device interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::SetMaterial

IDirect3DDevice7::GetRenderState

The **IDirect3DDevice7::GetRenderState** method gets a single Direct3DDevice rendering-state parameter.

```
HRESULT GetRenderState(  
    D3DRENDERSTATETYPE dwRenderStateType,  
    LPDWORD lpdwRenderState  
);
```

Parameters

dwRenderStateType

Device state variable that is being queried. This parameter can be any of the members of the **D3DRENDERSTATETYPE** enumerated type.

lpdwRenderState

Address of a variable that receives the Direct3DDevice render state when the method returns.

Return Values

If the method succeeds, the return value is **D3D_OK**.

If the method fails, the return value is an error. The method returns **DDERR_INVALIDPARAMS** if one of the arguments is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::SetRenderState

IDirect3DDevice7::GetRenderTarget

The **IDirect3DDevice7::GetRenderTarget** method retrieves a pointer to the DirectDraw surface that is being used as a render target.

```
HRESULT GetRenderTarget(  
    LPDIRECTDRAWSURFACE7 *lpRenderTarget  
);
```

Parameters

lpRenderTarget

Address that receives a pointer to the **IDirectDrawSurface7** interface of the render target surface for this device.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

In the **IDirect3DDevice3** interface, this method accepted a pointer to the legacy **IDirectDrawSurface4** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::SetRenderTarget

IDirect3DDevice7::GetStateData

The **IDirect3DDevice7::GetStateData** method is not implemented.

```
HRESULT GetStateData(  
    DWORD dwState,  
    LPVOID* lpStateData  
);
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3DDevice7::GetTexture

The **IDirect3DDevice7::GetTexture** method retrieves a texture assigned to a given stage for a device.

```
HRESULT GetTexture(  
    DWORD dwStage,  
    LPDIRECTDRAW_SURFACE7 * lpTexture  
);
```

Parameters

dwStage

Stage identifier of the texture to be retrieved. Stage identifiers are zero-based. Currently, devices can have up to eight set textures, so the maximum allowable value allowed for *dwStage* is 7.

lpTexture

Address of a variable to be filled with a pointer to the specified texture's **IDirectDrawSurface7** interface if the call 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:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

In the legacy **IDirect3DDevice3** interface, this method accepts a pointer to an **IDirect3DTexture2** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::SetTexture, **IDirect3DDevice7::GetTextureStageState**, **IDirect3DDevice7::SetTextureStageState**, Textures

IDirect3DDevice7::GetTextureStageState

The **IDirect3DDevice7::GetTextureStageState** method retrieves a state value for a currently assigned texture.

```
HRESULT GetTextureStageState(
    DWORD dwStage,
    D3DTEXTURESTAGESTATETYPE dwState,
    LPDWORD lpdwValue
);
```

Parameters

dwStage

Stage identifier of the texture for which the state is retrieved. Stage identifiers are zero-based. Currently, devices can have up to eight set textures, so the maximum allowable value allowed for *dwStage* is 7.

dwState

Texture state to be retrieved. This parameter can be any member of the **D3DTEXTURESTAGESTATETYPE** enumerated type.

lpdwValue

Address of a variable to be filled with the retrieved state value. The meaning of the retrieved value is determined by the *dwState* parameter.

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:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::SetTextureStageState, **IDirect3DDevice7::GetTexture**, **IDirect3DDevice7::SetTexture**, Textures

IDirect3DDevice7::GetTransform

The **IDirect3DDevice7::GetTransform** method gets a matrix describing a transformation state.

```
HRESULT GetTransform(  
    D3DTRANSFORMSTATETYPE dtstTransformStateType,  
    LPD3DMATRIX lpD3DMatrix  
);
```

Parameters

dtstTransformStateType

Device state variable that is being modified. This parameter can be any of the members of the **D3DTRANSFORMSTATETYPE** enumerated type.

lpD3DMatrix

Address of a **D3DMATRIX** structure describing the transformation.

Return Values

If the method succeeds, the return value is **D3D_OK**.

If the method fails, the return value is an error. The method returns **DDERR_INVALIDPARAMS** if one of the arguments is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in **d3d.h**.

See Also

IDirect3DDevice7::SetTransform

IDirect3DDevice7::GetViewport

The **IDirect3DDevice7::GetViewport** method retrieves the viewport parameters currently set for the device.


```
HRESULT GetViewport(  
    LPD3DVIEWPORT7 lpViewport  
);
```

Parameters

lpViewport

Address of a **D3DVIEWPORT7** structure to be filled with the current viewport parameters if the call succeeds.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if the *lpViewport* parameter is invalid.

Remarks

This method was introduced with the **IDirect3DDevice7** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::SetViewport

IDirect3DDevice7::LightEnable

The **IDirect3DDevice7::LightEnable** method enables or disables a set of lighting parameters within a device.

```
HRESULT LightEnable(  
    DWORD dwLightIndex,  
    BOOL bEnable  
);
```

Parameters

dwLightIndex

Zero-based index of the set of lighting parameters that are the target of this method.

bEnable

Value indicating 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 is an error. For a complete list of Direct3D Immediate Mode error values and descriptions, see Return Values.

Remarks

If you supply a value for *dwLightIndex* outside the range of the light property sets currently assigned within the device, the **LightEnable** method creates a light source with the following properties and sets its enabled state to the value specified in *bEnable*:

Member	Default
dltType	D3DLIGHT_DIRECTIONAL
dvcDiffuse	(R:1, G:1, B:1, A:0)
dvcSpecular	(R:0, G:0, B:0, A:0)
dvcAmbient	(R:0, G:0, B:0, A:0)
dvPosition	(0, 0, 0)
dvDirection	(0, 0, 1)
dvRange	0
dvFalloff	0
dvAttenuation0	0
dvAttenuation1	0
dvAttenuation2	0
dvTheta	0
dvPhi	0

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetLight, **IDirect3DDevice7::GetLightEnable**,
IDirect3DDevice7::SetLight, Lighting and Materials

IDirect3DDevice7::Load

The **IDirect3DDevice7::Load** method loads a rectangular area of a source texture to a specified point in a destination texture or to faces within a cubic environment map.

```
HRESULT Load(  
    LPDIRECTDRAWSURFACE7 lpDestTex,  
    LPPOINT lpDestPoint,  
    LPDIRECTDRAWSURFACE7 lpSrcTex,  
    LPRECT lpSrcRect,  
    DWORD dwFlags  
);
```

Parameters

lpDestTex

Address of the **IDirectDrawSurface7** interface of the texture that receives image data from the source texture. The destination texture can be a cubic environment map. For complex texture surfaces like cubic environment maps and mipmaps, this must point to the top-level surface.

lpDestPoint

Address of a **POINT** structure that describes the point at which the method loads the image data onto the destination texture. Set this parameter to NULL when the destination point should be the origin of the destination texture.

lpSrcTex

Address of the **IDirect3DTexture7** interface of the texture that contains the image data to be loaded. For complex texture surfaces like cubic environment maps and mipmaps, this must point to the top-level surface.

lpSrcRect

Address of a **RECT** structure that describes the area within the source texture that the method loads. Set this parameter to NULL when the source rectangle should cover the entire source texture.

dwFlags

Value indicating, in the case of a cubic environment map, which face of the destination texture is to receive the image data. This can be any combination of the following flags.

0

Required if the destination texture is a managed texture.

DDSCAPS2_CUBEMAP_ALLFACES

All faces should be loaded with the image data within the source texture.

DDSCAPS2_CUBEMAP_NEGATIVEX,
DDSCAPS2_CUBEMAP_NEGATIVEY, or
DDSCAPS2_CUBEMAP_NEGATIVEZ

The negative x, y, or z faces should receive the image data.

DDSCAPS2_CUBEMAP_POSITIVEX,
DDSCAPS2_CUBEMAP_POSITIVEY, or
DDSCAPS2_CUBEMAP_POSITIVEZ

The positive x, y, or z faces should receive the image data.

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:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

The destination texture can be used with any rendering device, not just the one that created it. On hardware devices, the load operation is hardware-accelerated.

Loading textures into video memory by calling this method is preferred over blit operations.

This method copies all mip-levels, cubemap faces, palettes and color keys from the source texture to the destination texture.

When using this method to load mipmaps, the following special points apply:

- The destination texture surface can be a subset of the mip-levels contained in the source texture. In this case, only the mip-levels common to both textures are copied. This is useful in implementing texture LOD. You can create a video memory texture with only two mip-levels, then use this method to copy those levels from a source texture that comprises the entire set of mip-levels.
- The source and destination surface pointers must point to top-level surfaces.
- When loading a subrectangle of a mipmap, the subrectangle refers to the top-level surface in the mipmap chain, and is divided by two for each lower mip-level.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3DDevice7::MultiplyTransform

The **IDirect3DDevice7::MultiplyTransform** method multiplies a device's world, view, or projection matrices by a specified matrix. The multiplication order is *lpD3DMatrix* times *dstTransformStateType*.

```
HRESULT MultiplyTransform(  
    D3DTRANSFORMSTATETYPE dstTransformStateType,  
    LPD3DMATRIX lpD3DMatrix  
);
```

Parameters

dstTransformStateType

A member of the **D3DTRANSFORMSTATETYPE** enumerated type that identifies which device matrix is to be modified. The most common setting, **D3DTRANSFORMSTATE_WORLD**, modifies the world matrix, but you can specify that the method modify the view or projection matrices, if needed.

lpD3DMatrix

Address of a **D3DMATRIX** structure that modifies the current transformation.

Return Values

If the method succeeds, the return value is **D3D_OK**.

If the method fails, the return value is an error. The method returns **DDERR_INVALIDPARAMS** if one of the arguments is invalid.

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 the parameters are shown in this pseudocode.)

```
IDirect3DDevice7::SetTransform(D3DTRANSFORMSTATE_WORLD,
```

```

    shoulder_transform)
IDirect3DDevice7::DrawPrimitive(upper_arm)
IDirect3DDevice7::MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
    elbow_transform)
IDirect3DDevice7::DrawPrimitive(lower_arm)
IDirect3DDevice7::MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
    wrist_transform)
IDirect3DDevice7::DrawPrimitive(hand)

```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::DrawPrimitive, **IDirect3DDevice7::SetTransform**

IDirect3DDevice7::PreLoad

The **IDirect3DDevice7::Preload** method instructs the Direct3D texture manager to load a managed texture into video memory.

```

HRESULT PreLoad(
    LPDIRECTDRAWSURFACE7 lpddsTexture
);

```

Parameters

lpddsTexture

Address of the **IDirectDrawSurface7** interface for the texture surface to be loaded into memory.

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:

```

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

```

Remarks

This method forces the system to load a managed texture into video memory. If sufficient video memory is not available, the system removes other textures to regain memory, then loads the texture. The texture being loaded must be a managed texture (created with the DDSCAPS2_TEXTUREMANAGE or DDSCAPS2_D3DTEXTUREMANAGE flags). If not, the method fails and returns DDERR_INVALIDPARAMS.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

Automatic Texture Management

IDirect3DDevice7::SetClipPlane

The **IDirect3DDevice7::SetClipPlane** method sets the coefficients of a user-defined clipping plane for the device.

```
HRESULT SetClipPlane(  
    DWORD dwIndex,  
    D3DVALUE* pPlaneEquation  
);
```

Parameters

dwIndex

Index of the clipping plane for which the plane equation coefficients are to be set.

pPlaneEquation

Address of 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 DDERR_INVALIDPARAMS. This error indicates that the value in *dwIndex* exceeds the maximum clipping plane index

supported by the device or that the array at *pPlaneEquation* 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 *pPlaneEquation* 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 $Ax + By + Cz + D = 0$. A point with homogeneous coordinates (x, y, z, w) is visible in the half space of the plane if $Ax + By + Cz + Dw \geq 0$. 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.

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 D3DRENDERSTATE_CLIPPLANEENABLE render state.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetClipPlane, D3DRENDERSTATE_CLIPPLANEENABLE, User-defined Clip Planes

IDirect3DDevice7::SetClipStatus

The **IDirect3DDevice7::SetClipStatus** method sets the current clip status.

```
HRESULT SetClipStatus(
    LPD3DCLIPSTATUS lpD3DClipStatus
);
```

Parameters

lpD3DClipStatus

Address of a **D3DCLIPSTATUS** structure that describes the new settings for the clip status.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetClipStatus

IDirect3DDevice7::SetLight

The **IDirect3DDevice7::SetLight** method assigns a set of lighting properties for this device.

```
HRESULT SetLight(  
    DWORD dwLightIndex,  
    LPD3DLIGHT7 lpLight  
);
```

Parameters

dwLightIndex

Zero-based index of the set of lighting properties to be set. If a set of lighting properties already exists at this index, it is overwritten by the new properties in *lpLight*.

lpLight

Address of a **D3DLIGHT7** structure that contains the lighting-parameters to be set.

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:

DDERR_INVALIDPARAMS

DDERR_OUTOFMEMORY

Remarks

This method was introduced with the **IDirect3DDevice7** interface.

Unlike its predecessors, the **IDirect3DDevice7** interface does not use light objects. This method, and its use of the **D3DLIGHT7** structure to describe a set of lighting properties, replaces the lighting semantics used by previous versions of the device interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetLight, **IDirect3DDevice7::GetLightEnable**,
IDirect3DDevice7::LightEnable

IDirect3DDevice7::SetMaterial

The **IDirect3DDevice7::SetMaterial** method sets the material properties for the device.

```
HRESULT SetMaterial(  
    LPD3DMATERIAL7 lpMaterial  
);
```

Parameters

lpMaterial

Address of a **D3DMATERIAL7** structure that describes the material properties to be set.

Return Values

If the method succeeds, the return value is **D3D_OK**.

If the method fails, the return value is an error. The method returns **DDERR_INVALIDPARAMS** if the *lpMaterial* parameter is invalid.

Remarks

This method was introduced with the **IDirect3DDevice7** interface.

Unlike its predecessors, the **IDirect3DDevice7** interface does not use material handles. This method, and its use of the **D3DMATERIAL7** structure to describe material properties, replaces the material-handle semantics used by previous versions of the device interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetMaterial

IDirect3DDevice7::SetRenderState

The **IDirect3DDevice7::SetRenderState** method sets a single Direct3DDevice render-state parameter.

```
HRESULT SetRenderState(  
    D3DRENDERSTATETYPE dwRenderStateType,  
    DWORD dwRenderState  
);
```

Parameters

dwRenderStateType

Device state variable that is being modified. This parameter can be any of the members of the **D3DRENDERSTATETYPE** enumerated type.

dwRenderState

New value for the Direct3DDevice render state. The meaning of this parameter is dependent on the value specified for *dwRenderStateType*. For example, if *dwRenderStateType* were **D3DRENDERSTATE_SHADEMODE**, the second parameter would be one of the members of the **D3DSHADEMODE** enumerated type.

Return Values

If the method succeeds, the return value is **D3D_OK**.

If the method fails, the return value is an error. The method returns `DDERR_INVALIDPARAMS` if one of the arguments is invalid.

Remarks

Applications should use the **IDirect3DDevice7::SetTextureStageState** method to set texture states, rather than the legacy texture-related render states.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Version: Requires DirectX 7.0.
Header: Declared in `d3d.h`.

See Also

IDirect3DDevice7::GetRenderState, **IDirect3DDevice7::SetTransform**, [About Render States](#)

IDirect3DDevice7::SetRenderTarget

The **IDirect3DDevice7::SetRenderTarget** method permits the application to easily route rendering output to a new DirectDraw surface as a render target.

```
HRESULT SetRenderTarget(  
    LPDIRECTDRAW_SURFACE7 lpNewRenderTarget,  
    DWORD dwFlags  
);
```

Parameters

lpNewRenderTarget

Address of a **IDirectDrawSurface7** interface for the surface object that is the new rendering target. This surface must be created with the `DDSCAPS_3DDEVICE` capability.

dwFlags

Not currently used; set to 0.

Return Values

If the method succeeds, the return value is `D3D_OK`.

If the method fails, the return value is an error. The error can be one of the following values:

DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE

Remarks

Do not use this method to set a new render target surface with a depth buffer if the current render target does not have a depth buffer. Likewise, you cannot use this method to switch from a nondepth-buffered render target to a depth-buffered render target. Attempts to do this fail in debug builds and can exhibit unreliable behavior in retail builds. Since both the new and the old render targets use depth buffers, the depth buffer attached to the new render target replaces the previous depth buffer for the context.

Unlike this method's implementation in previous interfaces, **IDirect3DDevice7::SetRenderTarget** does not invalidate the current material or viewport for the device.

Capabilities do not change with changes in the properties of the render target surface. Both the Direct3D hardware abstraction layer (HAL) and the software rasterizers have only one opportunity to expose capabilities to the application. The system cannot expose different sets of capabilities, depending on the format of the destination surface.

If more than one depth buffer is attached to the render target, this function fails.

In the **IDirect3DDevice3** interface, this method accepted a pointer to the legacy **IDirectDrawSurface4** interface.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Version: Requires DirectX 7.0.
Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetRenderTarget

IDirect3DDevice7::SetStateData

The **IDirect3DDevice7::SetStateData** method is not implemented.

```
HRESULT SetStateData(  
    DWORD dwState,  
    LPVOID lpStateData  
);
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

IDirect3DDevice7::SetTexture

The **IDirect3DDevice7::SetTexture** method assigns a texture to a given stage for a device.

```
HRESULT SetTexture(  
    DWORD dwStage,  
    LPDIRECTDRAWSURFACE7 lpTexture  
);
```

Parameters

dwStage

Stage identifier to which the texture is to be set. Stage identifiers are zero-based. Currently, devices can have up to eight set textures, so the maximum allowable value allowed for *dwStage* is 7.

lpTexture

Address of the **IDirectDrawSurface7** interface for the texture being set. For complex textures, such as mipmaps and cubic environment maps, 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 one of the following values:

```
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS
```

Remarks

This 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.

Software devices do not support assigning a texture to more than one texture stage at a time.

In the legacy **IDirect3DDevice3** interface, this method accepted a pointer to an **IDirect3DTexture2** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetTexture, **IDirect3DDevice7::GetTextureStageState**, **IDirect3DDevice7::SetTextureStageState**, Textures

IDirect3DDevice7::SetTextureStageState

The **IDirect3DDevice7::SetTextureStageState** method sets the state value for a currently assigned texture.

```
HRESULT SetTextureStageState(  
    DWORD dwStage,  
    D3DTEXTURESTAGESTATETYPE dwState,  
    DWORD dwValue  
);
```

Parameters

dwStage

Stage identifier of the texture for which the state value is to be set. Stage identifiers are zero-based. Currently, devices can have up to eight set textures, so the maximum allowable value allowed for *dwStage* is 7.

dwState

Texture state to be set. This parameter can be any member of the **D3DTEXTURESTAGESTATETYPE** enumerated type.

dwValue

State value to be set. The meaning of this value is determined by the *dwState* parameter.

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:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

Applications should use this method to set texture states, rather than the legacy texture-related render states. For more information, see [About Render States](#).

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Version: Requires DirectX 7.0.
Header: Declared in d3d.h.

See Also

[IDirect3DDevice7::GetTextureStageState](#), [IDirect3DDevice7::GetTexture](#), [IDirect3DDevice7::SetTexture](#), [Textures](#)

IDirect3DDevice7::SetTransform

The **IDirect3DDevice7::SetTransform** method sets a single Direct3DDevice transformation-related state.

```
HRESULT SetTransform(  
    D3DTRANSFORMSTATETYPE dtstTransformStateType,  
    LPD3DMATRIX lpD3DMatrix  
);
```

Parameters

dtstTransformStateType

Device-state variable that is being modified. This parameter can be any of the members of the **D3DTRANSFORMSTATETYPE** enumerated type.

lpD3DMatrix

Address of a **D3DMATRIX** structure that modifies the current transformation.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns `DDERR_INVALIDPARAMS` if one of the arguments is invalid.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in `d3d.h`.

See Also

`IDirect3DDevice7::GetTransform`, `IDirect3DDevice7::SetRenderState`

IDirect3DDevice7::SetViewport

The `IDirect3DDevice7::SetViewport` method sets the viewport parameters for the device.

```
HRESULT SetViewport(  
    LPD3DVIEWPORT7 lpViewport  
);
```

Parameters

lpViewport

Address of a `D3DVIEWPORT7` structure that contains the viewport parameters to be set.

Return Values

If the method succeeds, the return value is `D3D_OK`.

If the method fails, the return value is an error. The method returns `DDERR_INVALIDPARAMS` if the *lpViewport* parameter is invalid.

Remarks

This method was introduced with the `IDirect3DDevice7` interface.

If the viewport parameters described by the `D3DVIEWPORT7` structure describe a region that cannot exist within the render target surface, the method fails, returning `DDERR_INVALIDPARAMS`.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetViewport

IDirect3DDevice7::ValidateDevice

The **IDirect3DDevice7::ValidateDevice** method reports the device's ability to render the currently set texture-blending operations and arguments in a single pass.

```
HRESULT ValidateDevice(  
    LPDWORD lpdwPasses  
);
```

Parameters

lpdwPasses

Address to be filled with the number of rendering passes 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:

```
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS  
D3DERR_CONFLICTINGTEXTUREFILTER  
D3DERR_CONFLICTINGTEXTUREPALETTE  
D3DERR_TOOMANYOPERATIONS  
D3DERR_UNSUPPORTEDALPHAARG  
D3DERR_UNSUPPORTEDALPHAOPERATION  
D3DERR_UNSUPPORTEDCOLORARG  
D3DERR_UNSUPPORTEDCOLOROPERATION  
D3DERR_UNSUPPORTEDFACTORVALUE  
D3DERR_UNSUPPORTEDTEXTUREFILTER
```

D3DERR_WRONGTEXTUREFORMAT

Remarks

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 up the desired blending operation, then calling the **ValidateDevice** method.

The **ValidateDevice** method uses the currently set render states, textures, and texture-stage states to perform validation at the time of the call. Any 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 only introduce iterated color data 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 only available 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 both **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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DDevice7::GetTextureStageState,
IDirect3DDevice7::SetTextureStageState

IDirect3DVertexBuffer7

Applications use the methods of the **IDirect3DVertexBuffer7** interface to manipulate a collection of vertices for use with the **IDirect3DDevice7::DrawPrimitiveVB** and **IDirect3DDevice7::DrawIndexedPrimitiveVB** rendering methods. This section is a reference to the methods of this interface. For a conceptual overview, see Vertex Buffers.

The methods of the **IDirect3DVertexBuffer** interface can be organized into the following groups:

Information	GetVertexBufferDesc
Vertex data	Lock
	Optimize
	ProcessVertices
	ProcessVerticesStrided
	Unlock

The **IDirect3DVertexBuffer7** interface extends the **IDirect3DVertexBuffer** interface by adding the **IDirect3DVertexBuffer7::ProcessVerticesStrided** method, which supports the strided vertex format.

The **IDirect3DVertexBuffer7** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

The **LPDIRECT3DVERTEXBUFFER7** and **LPDIRECT3DVERTEXBUFFER** data types are defined as pointers to the **IDirect3DVertexBuffer7** and **IDirect3DVertexBuffer** interfaces:

```
typedef struct IDirect3DVertexBuffer *LPDIRECT3DVERTEXBUFFER;  

typedef struct IDirect3DVertexBuffer7 *LPDIRECT3DVERTEXBUFFER7;
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.
Version: Requires DirectX 7.0.
Header: Declared in d3d.h.

See Also

Vertex Buffers

IDirect3DVertexBuffer7::GetVertexBufferDesc

The **IDirect3DVertexBuffer7::GetVertexBufferDesc** method retrieves a description of the vertex buffer.

```
HRESULT GetVertexBufferDesc(  
    LPD3DVERTEXBUFFERDESC lpVBDesc  
);
```

Parameters

lpVBDesc

Address of a **D3DVERTEXBUFFERDESC** structure to be filled with a description of the vertex buffer.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be DDERR_INVALIDPARAMS or another error value.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Version: Requires DirectX 7.0.
Header: Declared in d3d.h.

IDirect3DVertexBuffer7::Lock

The **IDirect3DVertexBuffer7::Lock** methods locks a vertex buffer and obtains a pointer to the vertex buffer memory.

```
HRESULT Lock(  
    DWORD dwFlags,
```

```

LPVOID* lppData,
LPDWORD lpdwSize
);

```

Parameters

dwFlags

Flags indicating how the vertex buffer memory should be locked.

DDLOCK_EVENT

Not currently implemented.

DDLOCK_NOSYSLOCK

If possible, do not take the Win16Mutex (also known as Win16Lock).

DDLOCK_READONLY

Indicates that the memory being locked is only to be read from.

DDLOCK_SURFACEMEMORYPTR

Indicates that a valid memory pointer to the vertex buffer is returned; this is the default.

DDLOCK_WAIT

If a lock cannot be obtained immediately, the method retries until a lock is obtained or another error occurs.

DDLOCK_WRITEONLY

Indicates that the memory being locked is only to be written to.

DDLOCK_DISCARDCONTENTS

New for DirectX 7.0. Used only with Direct3D vertex-buffer locks. Indicates that no assumptions are made about the contents of the vertex buffer during this lock. This enables Direct3D or the driver to provide an alternative memory area as the vertex buffer. This is useful when you plan to clear the contents of the vertex buffer and fill in new data.

DDLOCK_NOOVERWRITE

New for DirectX 7.0. Used only with Direct3D vertex-buffer locks. Indicates that no vertices that were referred to in vertex-buffer DrawPrimitive calls since the start of the frame (or the last lock without this flag) are modified during the lock. This can be useful when you want only to append data to the vertex buffer.

lppData

Address of a variable that receives the address of the vertex buffer memory if the call succeeds.

lpdwSize

Address of a variable that receives the size of the vertex buffer memory at *lppData*. Set to NULL if the buffer size is not needed.

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_VERTEXBUFFEROPTIMIZED
 DDERR_INVALIDPARAMS
 DDERR_OUTOFMEMORY
 DDERR_SURFACEBUSY
 DDERR_SURFACELOST

Remarks

After locking the vertex buffer, you can access the memory until a corresponding call to **IDirect3DVertexBuffer7::Unlock**.

You cannot render from a locked vertex buffer; calls to the **IDirect3DDevice7::DrawIndexedPrimitiveVB** or **IDirect3DDevice7::DrawPrimitiveVB** method using a locked buffer fail, returning D3DERR_VERTEXBUFFERLOCKED.

This method often causes the system to hold the Win16Mutex until you call the **IDirect3DVertexBuffer7::Unlock** method. GUI debuggers cannot operate while the Win16Mutex is held.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DVertexBuffer7::Unlock, Accessing the Contents of a Vertex Buffer

IDirect3DVertexBuffer7::Optimize

The **IDirect3DVertexBuffer7::Optimize** method converts an unoptimized vertex buffer into an optimized vertex buffer.

```
HRESULT Optimize(
    LPDIRECT3DDEVICE7 lpD3DDevice,
    DWORD dwFlags
);
```

Parameters

lpD3DDevice

Address of the **IDirect3DDevice7** interface of the device for which this vertex buffer is to be optimized.

dwFlags

Not currently used; set to 0.

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_VERTEXBUFFEROPTIMIZED

D3DERR_VERTEXBUFFERLOCKED

DDERR_INVALIDPARAMS

DDERR_OUTOFMEMORY

Remarks

In the legacy **IDirect3DVertexBuffer** interface, this method accepted a pointer to the **IDirect3DDevice3** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

Optimizing a Vertex Buffer, Vertex Buffers

IDirect3DVertexBuffer7::ProcessVertices

The **IDirect3DVertexBuffer7::ProcessVertices** method processes untransformed vertices into a transformed or optimized vertex buffer.

HRESULT ProcessVertices(

DWORD *dwVertexOp*,

DWORD *dwDestIndex*,

DWORD *dwCount*,

LPDIRECT3DVERTEXBUFFER7 *lpSrcBuffer*,


```

DWORD dwSrcIndex,
LPDIRECT3DDEVICE7 lpD3DDevice,
DWORD dwFlags
);

```

Parameters

dwVertexOp

Flags defining how the method processes the vertices as they are transferred from the source buffer. You can specify any combination of the following flags:

D3DVOP_CLIP

Transform the vertices, and clip any vertices that exist outside the viewing frustum. This flag cannot be used with vertex buffers that do not contain clipping information (for example, those created with the D3DVBCAPS_DONOTCLIP flag).

D3DVOP_EXTENTS

Transform the vertices, then update the extents of the screen rectangle when the vertices are rendered. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice7::GetClipStatus** are not updated to account for the vertices when they are rendered.

D3DVOP_LIGHT

Light the vertices.

D3DVOP_TRANSFORM

Transform the vertices, using the world, view, and projection matrices. This flag must always be set.

dwDestIndex

Index into the destination vertex buffer (this buffer) where the vertices are placed after processing.

dwCount

Number of vertices in the source buffer to process.

lpSrcBuffer

Address of the **IDirect3DVertexBuffer7** interface for the source vertex buffer.

dwSrcIndex

Index of the first vertex in the source buffer to be processed.

lpD3DDevice

Address of the **IDirect3DDevice7** interface for the device to be used to transform the vertices.

dwFlags

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 current vertex operation into the destination buffer. For more information, see Processing Vertices.

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_INVALIDVERTEXFORMAT
 DDERR_INVALIDOBJECT
 DDERR_INVALIDPARAMS
 DDERR_OUTOFMEMORY
 DDERR_SURFACEBUSY
 DDERR_SURFACELOST

Remarks

Always include the D3DVOP_TRANSFORMED flag in the *dwVertexOp* parameter. If you do not, the method fails, returning DDERR_INVALIDPARAMS.

In the legacy **IDirect3DVertexBuffer** interface, this method accepted pointers to the **IDirect3DDevice3** and **IDirect3DVertexBuffer** interfaces.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Version: Requires DirectX 7.0.
Header: Declared in d3d.h.

See Also

Processing Vertices, Vertex Buffers

IDirect3DVertexBuffer7::ProcessVerticesStrided

The **IDirect3DVertexBuffer7::ProcessVerticesStrided** method processes untransformed strided vertices into a transformed or optimized vertex buffer.

```
HRESULT ProcessVerticesStrided(  

    DWORD dwVertexOp,  

    DWORD dwDestIndex,  

    DWORD dwCount,  

    LPD3DDRAWPRIMITIVESTRIDEDDATA lpVertexArray,  

    DWORD dwSrcIndex,  

    LPDIRECT3DDEVICE7 lpD3DDevice,
```

DWORD *dwFlags*
);

Parameters

dwVertexOp

Flags defining how the method processes the vertices as they are transferred from the source buffer. You can specify any combination of the following flags:

D3DVOP_CLIP

Transform the vertices, and clip any vertices that exist outside the viewing frustum. This flag cannot be used with vertex buffers that do not contain clipping information (for example, those created with the D3DVBCAPS_DONOTCLIP flag).

D3DVOP_EXTENTS

Transform the vertices, then update the extents of the screen rectangle when the vertices are rendered. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice7::GetClipStatus** are not updated to account for the vertices when they are rendered.

D3DVOP_LIGHT

Light the vertices.

D3DVOP_TRANSFORM

Transform the vertices, using the world, view, and projection matrices. This flag must always be set.

dwDestIndex

Index into the destination vertex buffer (this buffer) where the vertices are placed after processing.

dwCount

Number of vertices in the source buffer to process.

lpVertexArray

Array of **D3DDRAWPRIMITIVESTRIDEDDATA** structures that contains the vertices for this primitive.

dwSrcIndex

Index of the first vertex in the source buffer to be processed.

lpD3DDevice

Address of the **IDirect3DDevice7** interface for the device to be used to transform the vertices.

dwFlags

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 current vertex operation into the destination buffer. For more information, see Processing Vertices.

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_INVALIDVERTEXFORMAT
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_OUTOFMEMORY
- DDERR_SURFACEBUSY
- DDERR_SURFACELOST

Remarks

Always include the D3DVOP_TRANSFORMED flag in the *dwVertexOp* parameter. If you do not, the method fails, returning DDERR_INVALIDPARAMS.

This method was introduced with the **IDirect3DVertexBuffer7** interface.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Version: Requires DirectX 7.0.
Header: Declared in d3d.h.

See Also

IDirect3DVertexBuffer7::ProcessVertices, Processing Vertices, Vertex Buffers

IDirect3DVertexBuffer7::Unlock

The **IDirect3DVertexBuffer7::Unlock** method unlocks a previously locked vertex buffer.

```
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 one of the following values:

D3DERR_VERTEXBUFFERUNLOCKFAILED
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACEBUSY
DDERR_SURFACELOST

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3d.h.

See Also

IDirect3DVertexBuffer7::Lock, Accessing the Contents of a Vertex Buffer

Unimplemented Methods

The following methods were present in previous versions of DirectX, but are not implemented in the most recent versions of their interfaces. In some cases, entire interfaces were rendered obsolete. These interfaces are documented in previous versions of DirectX.

Former stub methods. These methods were never implemented and are not supported in any interface.

IDirect3D::Initialize

IDirect3DMaterial::Initialize

IDirect3DMaterial::Reserve

IDirect3DMaterial::Unreserve

IDirect3DTexture::Initialize

IDirect3DTexture::Unload

Obsolete methods. These methods are not supported in the most recent versions of their interfaces.

IDirect3D3::CreateLight

IDirect3D3::CreateMaterial

IDirect3D3::CreateViewport

IDirect3DDevice3::GetStats

IDirect3DDevice3::GetLightState**IDirect3DDevice3::SetLightState**

Superseded viewport methods. The following methods are superseded by the **IDirect3DDevice7::GetViewport** and **IDirect3DDevice7::SetViewport** methods.

IDirect3DDevice3::AddViewport**IDirect3DDevice3::DeleteViewport****IDirect3DDevice3::GetCurrentViewport****IDirect3DDevice3::NextViewport****IDirect3DDevice3::SetCurrentViewport**

Obsolete single-vertex rendering methods. The following methods are not supported in the **IDirect3DDevice7** interface.

IDirect3DDevice3::Begin**IDirect3DDevice3::BeginIndexed****IDirect3DDevice3::End****IDirect3DDevice3::Index****IDirect3DDevice3::Vertex**

Obsolete Interfaces

The **IDirect3DDevice** and **IDirect3DExecuteBuffer** interfaces that support rendering with execute buffers are obsolete. For information about these interfaces, see the documentation provided with a previous release of DirectX. Legacy documentation is provided with this SDK and available for download from <http://www.microsoft.com/directx>.

The **IDirect3DLight**, **IDirect3DMaterial3** and **IDirect3DViewport3** interfaces are not supported by interfaces more recent than the **IDirect3DDevice3** interface. The functionality offered by the superseded interfaces is now offered within the device interface itself (**IDirect3DDevice7**).

The **IDirect3DLight** interface is obsolete. The functionality provided by the interface is now present within the **IDirect3DDevice7** interface. To summarize the relationship between the two interfaces:

IDirect3DLight::GetLight and **IDirect3DLight::GetLight**

Superseded by the **IDirect3DDevice7::GetLight** and
IDirect3DDevice7::SetLight methods.

IDirect3DLight::Initialize

Not supported.

The following are the methods of the obsolete **IDirect3DMaterial3** interface. Where appropriate, correlations have been drawn to methods of the **IDirect3DDevice7** interface.

IDirect3DMaterial3::GetHandle

Not supported. Material handles are not used by the **IDirect3DDevice7** interface.

IDirect3DMaterial3::GetMaterial and **IDirect3DMaterial3::SetMaterial**

Superseded by the **IDirect3DDevice7::GetMaterial** and **IDirect3DDevice7::SetMaterial** methods.

The following list contains methods from the obsolete **IDirect3DViewport3** interface. Similar to the material-related methods in the preceding list, equivalent methods within **IDirect3DDevice7** are noted.

IDirect3DViewport3::GetBackground and**IDirect3DViewport3::SetBackground**

Not supported. Background materials are not used by the **IDirect3DDevice7** interface.

IDirect3DViewport3::GetBackgroundDepth,**IDirect3DViewport3::GetBackgroundDepth2,****IDirect3DViewport3::SetBackgroundDepth,** and**IDirect3DViewport3::SetBackgroundDepth2**

Not supported.

IDirect3DViewport3::AddLight, IDirect3DViewport3::NextLight, and**IDirect3DViewport3::DeleteLight**

Parameters for individual lights are accessed through the **IDirect3DDevice7::GetLight** and **IDirect3DDevice7::SetLight** methods.

Lights are individually enabled and disabled through the

IDirect3DDevice7::GetLightEnable and **IDirect3DDevice7::LightEnable** methods.

IDirect3DViewport3::LightElements

Not supported.

IDirect3DViewport3::Clear and **IDirect3DViewport3::Clear2**

Superseded by the **IDirect3DDevice7::Clear** method.

IDirect3DViewport3::GetViewport, IDirect3DViewport3::GetViewport2,**IDirect3DViewport3::SetViewport,** and **IDirect3DViewport3::SetViewport2**

Superseded by the **IDirect3DDevice7::GetViewport** and

IDirect3DDevice7::SetViewport methods.

IDirect3DViewport3::Initialize

Not supported.

IDirect3DViewport3::TransformVertices

Not supported. Use vertex buffers to manually transform vertices.

D3D_OVERLOADS

C++ programmers who define **D3D_OVERLOADS** can use the extensions documented here to simplify their code in Direct3D Immediate Mode applications.

The use of D3D_OVERLOADS was introduced with Microsoft® DirectX® 5.0. This section is a reference to the D3D_OVERLOADS extensions.

These extensions must be defined with C++ linkage. If D3D_OVERLOADS is defined and the inclusion of D3dtypes.h or D3d.h is surrounded by extern "C", link errors result. For example, the following syntax would generate link errors because of C linkage of D3D_OVERLOADS functionality:

```
#define D3D_OVERLOADS
extern "C" {
#include <d3d.h>
};
```

The D3D_OVERLOADS extensions can be organized into the following groups:

Constructors	D3DLVERTEX
	D3DTLVERTEX
	D3DVECTOR
	D3DVERTEX
Operators	Access Grant Operators
	Addition Operator
	Assignment Operators
	Bitwise Equality Operator
	D3DMATRIX
	Division Operator
	Multiplication Operator
	Subtraction Operator
	Unary Operators
Helper functions	Vector Dominance Operators
	CrossProduct
	DotProduct
	Magnitude
	Max
	Maximize
	Min
	Minimize
	Normalize
	SquareMagnitude

D3D_OVERLOADS Constructors

This section contains reference information for the constructors provided by the D3D_OVERLOADS C++ extensions.

- D3DLVERTEX
- D3DTLVERTEX
- D3DVECTOR
- D3DVERTEX

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

D3DLVERTEX Constructors

The D3D_OVERLOADS constructors for the **D3DLVERTEX** structure offer a convenient way for C++ programmers to create lit vertices.

```
_D3DLVERTEX() {}
_D3DLVERTEX(const D3DVECTOR& v,
             D3DCOLOR _color, D3DCOLOR _specular,
             float _tu, float _tv)
{ x = v.x; y = v.y; z = v.z; dwReserved = 0;
  color = _color; specular = _specular;
  tu = _tu; tv = _tv;
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

D3DTLVERTEX Constructors

The D3D_OVERLOADS constructors for the **D3DTLVERTEX** structure offer a convenient way for C++ programmers to create transformed and lit vertices.

```

_D3DTLVERTEX() {}
_D3DTLVERTEX(const D3DVECTOR& v, float _rhw,
              D3DCOLOR _color, D3DCOLOR _specular,
              float _tu, float _tv)
{
    sx = v.x; sy = v.y; sz = v.z; rhw = _rhw;
    color = _color; specular = _specular;
    tu = _tu; tv = _tv;
}

```

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

D3DVECTOR Constructors

The D3D_OVERLOADS constructors for the **D3DVECTOR** structure offer a convenient way for C++ programmers to create vectors.

```

_D3DVECTOR() {}
_D3DVECTOR(D3DVALUE f);
_D3DVECTOR(D3DVALUE _x, D3DVALUE _y, D3DVALUE _z);
_D3DVECTOR(const D3DVALUE f[3]);

```

These constructors are defined as follows:

```

inline _D3DVECTOR::_D3DVECTOR(D3DVALUE f)
{
    x = y = z = f;
}

inline _D3DVECTOR::_D3DVECTOR(D3DVALUE _x, D3DVALUE _y, D3DVALUE _z)
{
    x = _x; y = _y; z = _z;
}

inline _D3DVECTOR::_D3DVECTOR(const D3DVALUE f[3])
{
    x = f[0]; y = f[1]; z = f[2];
}

```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

D3DVERTEX Constructors

The D3D_OVERLOADS constructors for the **D3DVERTEX** structure offer a convenient way for C++ programmers to create lit vertices.

```
_D3DVERTEX() { }
_D3DVERTEX(const D3DVECTOR& v, const D3DVECTOR& n, float _tu, float _tv)
{ x = v.x; y = v.y; z = v.z;
  nx = n.x; ny = n.y; nz = n.z;
  tu = _tu; tv = _tv;
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

D3D_OVERLOADS Operators

This section contains reference information for the operators provided by the D3D_OVERLOADS C++ extensions.

- Access Grant Operators
- Addition Operator
- Assignment Operators
- Bitwise Equality Operator
- D3DMATRIX
- Division Operator
- Multiplication Operator
- Subtraction Operator
- Unary Operators
- Vector Dominance Operators

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

Access Grant Operators (D3D_ OVERLOADS)

The bracket ("`[]`") operators are overloaded operators for the D3D_ OVERLOADS extensions. You can use empty brackets ("`[]`") for access grants, "`v[0]`" to access the x component of a vector, "`v[1]`" to access the y component, and "`v[2]`" to access the z component. These operators are defined as follows:

```
const D3DVALUE&operator[ ](int i) const;
D3DVALUE&operator[ ](int i);
```

```
inline const D3DVALUE&
_D3DVECTOR::operator[ ](int i) const
{
    return (&x)[i];
}
```

```
inline D3DVALUE&
_D3DVECTOR::operator[ ](int i)
{
    return (&x)[i];
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

Addition Operator (D3D_ OVERLOADS)

This binary operator is an overloaded operator for the D3D_ OVERLOADS extensions. The addition operator is defined as follows:

```
_D3DVECTOR operator + (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

```
inline _D3DVECTOR
operator + (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
```

```
{
    return _D3DVECTOR(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

Assignment Operators (D3D_OVERLOADS)

The assignment operators are overloaded operators for the D3D_OVERLOADS extensions. Both scalar and vector forms of the "*" and "/" operators are implemented. (In the vector form, multiplication and division are memberwise.)

```
_D3DVECTOR& operator += (const _D3DVECTOR& v);
_D3DVECTOR& operator -= (const _D3DVECTOR& v);
_D3DVECTOR& operator *= (const _D3DVECTOR& v);
_D3DVECTOR& operator /= (const _D3DVECTOR& v);
_D3DVECTOR& operator *= (D3DVALUE s);
_D3DVECTOR& operator /= (D3DVALUE s);
```

The assignment operators are defined as follows:

```
inline _D3DVECTOR&
_D3DVECTOR::operator += (const _D3DVECTOR& v)
{
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

```
inline _D3DVECTOR&
_D3DVECTOR::operator -= (const _D3DVECTOR& v)
{
    x -= v.x; y -= v.y; z -= v.z;
    return *this;
}
```

```
inline _D3DVECTOR&
_D3DVECTOR::operator *= (const _D3DVECTOR& v)
{

```

```

    x *= v.x; y *= v.y; z *= v.z;
    return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator /= (const _D3DVECTOR& v)
{
    x /= v.x; y /= v.y; z /= v.z;
    return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator *= (D3DVALUE s)
{
    x *= s; y *= s; z *= s;
    return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator /= (D3DVALUE s)
{
    x /= s; y /= s; z /= s;
    return *this;
}

```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

Bitwise Equality Operator (D3D_OVERLOADS)

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. The bitwise equality operator is defined as follows:

```

int operator == (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline int
operator == (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{

```

```

return v1.x==v2.x && v1.y==v2.y && v1.z == v2.z;
}

```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

D3DMATRIX (D3D_OVERLOADS)

The D3D_OVERLOADS implementation of the **D3DMATRIX** structure 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, C++ programmers can refer to them by row and column number and index these numbers, as needed.

```

typedef struct _D3DMATRIX {
#ifdef __cplusplus && (defined D3D_OVERLOADS)
    union {
        struct {
D3DVALUE    _11, _12, _13, _14;
D3DVALUE    _21, _22, _23, _24;
D3DVALUE    _31, _32, _33, _34;
D3DVALUE    _41, _42, _43, _44;

#ifdef (defined __cplusplus && (defined D3D_OVERLOADS))
        };
        D3DVALUE m[4][4];
        };
        _D3DMATRIX() {}

        D3DVALUE& operator()(int iRow, int iColumn) { return m[iRow][iColumn]; }
        const D3DVALUE& operator()(int iRow, int iColumn) const { return m[iRow][iColumn]; }
#endif
    } D3DMATRIX, *LPD3DMATRIX;

```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

See Also

D3DMATRIX

Division Operator (D3D_OVERLOADS)

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. Both scalar and vector forms of this operator are implemented. The division operator is defined as follows:

```
_D3DVECTOR operator / (const _D3DVECTOR& v, D3DVALUE s);
_D3DVECTOR operator / (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

```
inline _D3DVECTOR
operator / (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
    return _D3DVECTOR(v1.x/v2.x, v1.y/v2.y, v1.z/v2.z);
}
```

```
inline _D3DVECTOR
operator / (const _D3DVECTOR& v, D3DVALUE s)
{
    return _D3DVECTOR(v.x/s, v.y/s, v.z/s);
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

Multiplication Operator (D3D_OVERLOADS)

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. Both scalar and vector forms of this operator are implemented. The multiplication operator is defined as follows:


```

_D3DVECTOR operator * (const _D3DVECTOR& v, D3DVALUE s);
_D3DVECTOR operator * (D3DVALUE s, const _D3DVECTOR& v);
_D3DVECTOR operator * (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

```

```

inline _D3DVECTOR
operator * (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
    return _D3DVECTOR(v1.x*v2.x, v1.y*v2.y, v1.z*v2.z);
}

```

```

inline _D3DVECTOR
operator * (const _D3DVECTOR& v, D3DVALUE s)
{
    return _D3DVECTOR(s*v.x, s*v.y, s*v.z);
}

```

```

inline _D3DVECTOR
operator * (D3DVALUE s, const _D3DVECTOR& v)
{
    return _D3DVECTOR(s*v.x, s*v.y, s*v.z);
}

```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

Subtraction Operator (D3D__OVERLOADS)

This binary operator is an overloaded operator for the D3D__OVERLOADS extensions. The subtraction operator is defined as follows:

```

_D3DVECTOR operator - (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

```

```

inline _D3DVECTOR
operator - (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
    return _D3DVECTOR(v1.x-v2.x, v1.y-v2.y, v1.z-v2.z);
}

```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3dtypes.h.

Unary Operators (D3D_OVERLOADS)

The unary operators are overloaded operators for the D3D_OVERLOADS extensions. The unary operators are defined as follows:

```
_D3DVECTOR operator + (const _D3DVECTOR& v);
_D3DVECTOR operator - (const _D3DVECTOR& v);
```

```
inline _D3DVECTOR
operator + (const _D3DVECTOR& v)
{
    return v;
}
```

```
inline _D3DVECTOR
operator - (const _D3DVECTOR& v)
{
    return _D3DVECTOR(-v.x, -v.y, -v.z);
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3dtypes.h.

Vector Dominance Operators (D3D_OVERLOADS)

These binary operators are overloaded operators for the D3D_OVERLOADS extensions. Vector v1 dominates vector v2 if any component of v1 is greater than the corresponding component of v2. Therefore, it is possible for neither of the two specified vectors to dominate the other.

```
int operator < (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

```
int operator <= (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

The vector-dominance operators are defined as follows:

```
inline int
operator < (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
    return v1[0] < v2[0] && v1[1] < v2[1] && v1[2] < v2[2];
}
```

```
inline int
operator <= (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
    return v1[0] <= v2[0] && v1[1] <= v2[1] && v1[2] <= v2[2];
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3dtypes.h.

D3D_OVERLOADS Helper Functions

This section contains reference information for the helper functions provided by the D3D_OVERLOADS C++ extensions.

- **CrossProduct**
- **DotProduct**
- **Magnitude**
- **Max**
- **Maximize**
- **Min**
- **Minimize**
- **Normalize**
- **SquareMagnitude**

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3dtypes.h.

CrossProduct

This helper function returns the cross product of the specified vectors. **CrossProduct** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR CrossProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline _D3DVECTOR
CrossProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
    _D3DVECTOR result;

    result[0] = v1[1] * v2[2] - v1[2] * v2[1];
    result[1] = v1[2] * v2[0] - v1[0] * v2[2];
    result[2] = v1[0] * v2[1] - v1[1] * v2[0];

    return result;
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

See Also

DotProduct

DotProduct

This helper function returns the dot product of the specified vectors. **DotProduct** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE DotProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline D3DVALUE  
DotProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2)  
{  
    return v1.x*v2.x + v1.y * v2.y + v1.z*v2.z;  
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

See Also

CrossProduct

Magnitude

This helper function returns the absolute value of the specified vector. **Magnitude** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE Magnitude (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE  
Magnitude (const _D3DVECTOR& v)  
{  
    return (D3DVALUE) sqrt(SquareMagnitude(v));  
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

See Also

SquareMagnitude

Max

This helper function returns the maximum component of the specified vector. **Max** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE Max (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE  
Max (const _D3DVECTOR& v)  
{  
    D3DVALUE ret = v.x;  
    if (ret < v.y) ret = v.y;  
    if (ret < v.z) ret = v.z;  
    return ret;  
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

See Also

Min

Maximize

This helper function returns a vector made up of the largest components of the two specified vectors. **Maximize** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR Maximize (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline _D3DVECTOR
Maximize (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
    return _D3DVECTOR( v1[0] > v2[0] ? v1[0] : v2[0],
                      v1[1] > v2[1] ? v1[1] : v2[1],
                      v1[2] > v2[2] ? v1[2] : v2[2]);
}
```

Remarks

You could use the **Maximize** and **Minimize** functions to compute the bounding box for a set of points, in a function that looks like this:

```
void
ComputeBoundingBox(const D3DVECTOR *pts, int N, D3DVECTOR *min, D3DVECTOR
*max)
{
    int i;
    *min = *max = pts[0];
    for (i = 1; i < N; i += 1)
    {
        *min = Minimize(*min, pts[i]);
        *max = Maximize(*max, pts[i]);
    }
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

See Also

Minimize

Min

This helper function returns the minimum component of the specified vector. **Min** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE Min (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE  
Min (const _D3DVECTOR& v)  
{  
    D3DVALUE ret = v.x;  
    if (v.y < ret) ret = v.y;  
    if (v.z < ret) ret = v.z;  
    return ret;  
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

See Also

Max

Minimize

This helper function returns a vector that is made up of the smallest components of the two specified vectors. **Minimize** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR Minimize (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline _D3DVECTOR  
Minimize (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
```



```
{
    return _D3DVECTOR( v1[0] < v2[0] ? v1[0] : v2[0],
                      v1[1] < v2[1] ? v1[1] : v2[1],
                      v1[2] < v2[2] ? v1[2] : v2[2]);
}
```

Remarks

You can use the **Maximize** and **Minimize** functions to compute the bounding box for a set of points, in a function that looks like this:

```
void
ComputeBoundingBox(const D3DVECTOR *pts, int N, D3DVECTOR *min, D3DVECTOR
*max)
{
    int i;
    *min = *max = pts[0];
    for (i = 1; i < N; i += 1)
    {
        *min = Minimize(*min, pts[i]);
        *max = Maximize(*max, pts[i]);
    }
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

See Also

Maximize

Normalize

This helper function returns the normalized version of the specified vector (that is, a unit-length vector with the same direction as the source). **Normalize** is part of the suite of extra functionality that is available to C++ programmers who define **D3D_OVERLOADS**.

```
_D3DVECTOR Normalize (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline _D3DVECTOR  
Normalize (const _D3DVECTOR& v)  
{  
    return v / Magnitude(v);  
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

SquareMagnitude

This helper function returns the square of the absolute value of the specified vector. **SquareMagnitude** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

D3DVALUE SquareMagnitude (const _D3DVECTOR& v);

This function is defined as follows:

```
inline D3DVALUE  
SquareMagnitude (const _D3DVECTOR& v)  
{  
    return v.x*v.x + v.y*v.y + v.z*v.z;  
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3d.h.

See Also

Magnitude

Callback Functions

This section contains reference information for the callback functions that you might need to implement when you work with Direct3D Immediate Mode. The following functions are implemented:

- **D3DEnumDevicesCallback7**
- **D3DEnumPixelFormatsCallback**

D3DEnumDevicesCallback7

The **D3DEnumDevicesCallback7** is an application-defined callback function for the **IDirect3D7::EnumDevices** method.

```
HRESULT CALLBACK D3DEnumDevicesCallback7(
    LPSTR lpDeviceDescription,
    LPSTR lpDeviceName,
    LPD3DDEVICEDESC7 lpD3DDeviceDesc,
    LPVOID lpContext
);
```

Parameters

lpDeviceDescription

Address of a textual description of the device.

lpDeviceName

Address of the device name.

lpD3DDeviceDesc

Address of a **D3DDEVICEDESC7** structure that contains the hardware capabilities of the Direct3D device.

lpContext

Address of application-defined data passed to this callback function.

Return Values

Applications should return **D3DENUMRET_OK** to continue the enumeration, or **D3DENUMRET_CANCEL** to cancel it.

Remarks

This callback function supersedes the **D3DEnumDevicesCallback** callback for all interfaces later than **IDirect3D3**.

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

The **LPD3DENUMDEVICESCALLBACK7** data type is defined as a pointer to this callback function:

```
typedef HRESULT (CALLBACK * LPD3DENUMDEVICESCALLBACK7)(
    LPSTR lpDeviceDescription, LPSTR lpDeviceName,
    LPD3DDEVICEDESC7, LPVOID);
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3dcaps.h.

D3DEnumPixelFormatCallback

The **D3DEnumPixelFormatCallback** is an application-defined callback function for the **IDirect3D3::EnumZBufferFormats**, and **IDirect3DDevice3::EnumTextureFormats** methods.

```
HRESULT CALLBACK D3DEnumPixelFormatCallback(
    LPDDPIXELFORMAT lpDDPixFmt,
    LPVOID lpContext
);
```

Parameters

lpDDPixFmt

Address of a **DDPIXELFORMAT** structure that describes the enumerated pixel format.

lpContext

Address of application-defined data passed to the function by the caller.

Return Values

Applications should return **D3DENUMRET_OK** to continue the enumeration, or **D3DENUMRET_CANCEL** to cancel it.

Remarks

The **LPD3DENUMPIXELFORMATSCALLBACK** data type is defined as a pointer to this callback function:

```
typedef HRESULT (WINAPI* LPD3DENUMPIXELFORMATSCALLBACK)
(LPDDPIXELFORMAT lpDDPixFmt, LPVOID lpContext);
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dcaps.h.

Obsolete Callback Functions

The Direct3D header files declare some callback functions for backward compatibility that are obsolete or have been superseded by new callback functions. Although obsolete, documentation on these callback function is available through previous releases of DirectX. Legacy documentation is included with this SDK and is available for download from <http://www.microsoft.com/directx>.

D3DEnumDevicesCallback

Obsolete and superseded by the **D3DEnumDevicesCallback7** callback.

D3DEnumTextureFormatsCallback

Obsolete and superseded by the **D3DEnumPixelFormatCallback** callback.

D3DValidateCallback

Obsolete. Information about this function, and other topics related to execute-buffer rendering, is available with the documentation provided for previous releases of DirectX.

Macros

This section contains reference information for the macros provided by Direct3D Immediate Mode.

- **D3DCLIPPLANE n**
- **D3DDivide**
- **D3DFVF_TEXCOORDSIZE n**
- **D3DMultiply**
- **D3DRGB**
- **D3DRGBA**
- **D3DVAL**
- **D3DVALP**
- **RGB_GETBLUE**
- **RGB_GETGREEN**
- **RGB_GETRED**
- **RGB_MAKE**
- **RGB_TORGBA**

- **RGBA_GETALPHA**
- **RGBA_GETBLUE**
- **RGBA_GETGREEN**
- **RGBA_GETRED**
- **RGBA_MAKE**
- **RGBA_SETALPHA**
- **RGBA_TORGB**

D3DCLIPPLANE n

The **D3DCLIPPLANE n** macros define bit patterns that enable user-defined clipping planes. These macros are defined as a convenience when setting values for the **D3DRENDERSTATE_CLIPPLANEENABLE** render state.

```
#define D3DCLIPPLANE0 (1 << 0)
#define D3DCLIPPLANE1 (1 << 1)
#define D3DCLIPPLANE2 (1 << 2)
#define D3DCLIPPLANE3 (1 << 3)
#define D3DCLIPPLANE4 (1 << 4)
#define D3DCLIPPLANE5 (1 << 5)
#define D3DCLIPPLANE6 (1 << 6)
#define D3DCLIPPLANE7 (1 << 7)
```

Parameters

None.

Remarks

User-defined clipping planes are enabled when the **DWORD** value set in the **D3DRENDERSTATE_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 planes 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.

See Also

D3DRENDERSTATE_CLIPPLANEENABLE

D3DDivide

The **D3DDivide** macro divides two values.

D3DDivide(a, b) (float)((double) (a) / (double) (b))

Parameters

a and *b*

Dividend and divisor in the expression, respectively.

Return Values

Returns the quotient of the division.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DMultiply

D3DFVF_TEXCOORDSIZE*n*

The **D3DFVF_TEXCOORDSIZE*n*** macros construct 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 identifying the texture coordinate set at which the texture coordinate size (1-, 2-, 3-, or 4-dimensional) applies.

Remarks

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3dtypes.h.

D3DMultiply

The **D3DMultiply** macro multiplies two values.

D3DMultiply(a, b) ((a) * (b))

Parameters

a and *b*

Values to be multiplied.

Return Values

Returns the product of the multiplication.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.
Version: Requires DirectX 2.0 or later.
Header: Declared in d3dtypes.h.

See Also

D3DDivide

D3DRGB

The **D3DRGB** macro initializes a color with the supplied RGB values.

```
D3DRGB(r, g, b) \
(0xff000000L | ( ((long)((r) * 255)) << 16) | \
(((long)((g) * 255)) << 8) | (long)((b) * 255))
```

Parameters

r, *g*, and *b*

Red, green, and blue components of the color. These should be floating-point values in the range 0 through 1.

Return Values

Returns the **D3DCOLOR** value corresponding to the supplied RGB values.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Version: Requires DirectX 2.0 or later.
Header: Declared in d3dtypes.h.

See Also

D3DRGBA

D3DRGBA

The **D3DRGBA** macro initializes a color with the supplied RGBA values.

```
D3DRGBA(r, g, b, a) \
((((long)((a) * 255)) << 24) |
(((long)((r) * 255)) << 16) |
(((long)((g) * 255)) << 8) | (long)((b) * 255))
```

Parameters

r, *g*, *b*, and *a*

Red, green, blue, and alpha components of the color.

Return Values

Returns the **D3DCOLOR** value corresponding to the supplied RGBA values.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DRGB

D3DVAL

The **D3DVAL** macro creates a value whose type is **D3DVALUE**.

D3DVAL(val) ((float)val)

Parameters

val

Value to be converted.

Return Values

Returns the converted value.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DVALP

D3DVALP

The **D3DVALP** macro creates a value of the specified precision.

D3DVALP(val, prec) ((float)val)

Parameters

val

Value to be converted.

prec

Ignored.

Return Values

Returns the converted value.

Remarks

The precision, as implemented by the **D3DVAL** macro, is 16 bits for the fractional part of the value.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DVAL

RGB_GETBLUE

The **RGB_GETBLUE** macro retrieves the blue component of a **D3DCOLOR** value.

RGB_GETBLUE(rgb) ((rgb) & 0xff)

Parameters

rgb

Color index from which the blue component is retrieved.

Return Values

Returns the blue component.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGB_GETGREEN

The **RGB_GETGREEN** macro retrieves the green component of a **D3DCOLOR** value.

RGB_GETGREEN(*rgb*) (((*rgb*) >> 8) & 0xff)

Parameters

rgb

Color index from which the green component is retrieved.

Return Values

Returns the green component.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGB_GETRED

The **RGB_GETRED** macro retrieves the red component of a **D3DCOLOR** value.

```
RGB_GETRED(rgb)  (((rgb) >> 16) & 0xff)
```

Parameters

rgb

Color index from which the red component is retrieved.

Return Values

Returns the red component.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGB_MAKE

The **RGB_MAKE** macro creates an RGB color from supplied values.

```
RGB_MAKE(r, g, b)  ((D3DCOLOR) (((r) << 16) |  
((g) << 8) | (b)))
```

Parameters

r, *g*, and *b*

Red, green, and blue components of the color to be created. These should be integer values in the range 0 through 255.

Return Values

Returns the color.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGB_TORGBA

The **RGB_TORGBA** macro creates an RGBA color from a supplied RGB color.

RGB_TORGBA(rgb) ((D3DCOLOR) ((rgb) | 0xff000000))

Parameters

rgb

RGB color to be converted to an RGBA color.

Return Values

Returns the RGBA color.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

RGBA_TORGB

RGBA_GETALPHA

The **RGBA_GETALPHA** macro retrieves the alpha component of an RGBA **D3DCOLOR** value.

RGBA_GETALPHA(rgb) ((rgb) >> 24)

Parameters

rgb

Color index from which the alpha component is retrieved.

Return Values

Returns the alpha component.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGBA_GETBLUE

The **RGBA_GETBLUE** macro retrieves the blue component of an **RGBA_D3DCOLOR** value.

RGBA_GETBLUE(rgb) ((rgb) & 0xff)

Parameters

rgb

Color index from which the blue component is retrieved.

Return Values

Returns the blue component.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGBA_GETGREEN

The **RGBA_GETGREEN** macro retrieves the green component of an **RGBA_D3DCOLOR** value.

RGBA_GETGREEN(rgb) (((rgb) >> 8) & 0xff)

Parameters

rgb

Color index from which the green component is retrieved.

Return Values

Returns the green component.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGBA_GETRED

The **RGBA_GETRED** macro retrieves the red component of an **RGBA D3DCOLOR** value.

```
RGBA_GETRED(rgb)  (((rgb) >> 16) & 0xff)
```

Parameters

rgb

Color index from which the red component is retrieved.

Return Values

Returns the red component.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGBA_MAKE

The **RGBA_MAKE** macro creates an **RGBA D3DCOLOR** value from supplied red, green, blue, and alpha components.

```
RGBA_MAKE(r, g, b, a)  \
((D3DCOLOR) (((a) << 24) | ((r) << 16) | ((g) << 8) | (b)))
```


Parameters

r, *g*, *b*, and *a*

Red, green, blue, and alpha components of the RGBA color to be created.

Return Values

Returns the color.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGBA_SETALPHA

The **RGBA_SETALPHA** macro sets the alpha component of an **RGBA_D3DCOLOR** value.

```
RGBA_SETALPHA(rgba, x) (((x) << 24) | ((rgba) & 0x00ffffff))
```

Parameters

rgba

RGBA color for which the alpha component is to be set.

x

Value of alpha component to be set.

Return Values

Returns the RGBA color whose alpha component has been set.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

RGBA_TORGB

The **RGBA_TORGB** macro creates an RGB **D3DCOLOR** value from a supplied RGBA **D3DCOLOR** value by stripping off the alpha component of the color.

RGBA_TORGB(rgba) ((D3DCOLOR) ((rgba) & 0xfffff))

Parameters

rgba

RGBA color to be converted to an RGB color.

Return Values

Returns the RGB color.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

RGB_TORGBA

Obsolete Macros

The **D3DSTATE_OVERRIDE** macro was used with the obsolete execute-buffer rendering architecture. Although obsolete, the Direct3D header files declare the macro for backward compatibility. For documentation about this macro and other execute-buffer topics, see the documentation provided with a previous release of DirectX. Legacy documentation is provided with this SDK and is available for download from <http://www.microsoft.com/directx>.

Structures

This section contains information about the following structures used with Direct3D Immediate Mode.

- **D3DCLIPSTATUS**
- **D3DCOLORVALUE**

- **D3DDEVICEDESC7**
- **D3DDEVINFO_TEXTUREMANAGER**
- **D3DDEVINFO_TEXTUREING**
- **D3DDP_PTRSTRIDE**
- **D3DDRAWPRIMITIVESTRIDEDDATA**
- **D3DLIGHT7**
- **D3DLIGHTINGCAPS**
- **D3DLINEPATTERN**
- **D3DLVERTEX**
- **D3DMATERIAL7**
- **D3DMATRIX**
- **D3DPRIMCAPS**
- **D3DRECT**
- **D3DTLVERTEX**
- **D3DVECTOR**
- **D3DVERTEX**
- **D3DVERTEXBUFFERDESC**
- **D3DVIEWPORT7**

Note

The memory for all DirectX structures must be initialized to 0 before use. In addition, all structures that contain a **dwSize** member must set the member to the size of the structure, in bytes, before use. The following example performs these tasks on a common structure, **DDCAPS**:

```
DDCAPS ddcaps; // Can't use this yet.

ZeroMemory(&ddcaps, sizeof(ddcaps));
ddcaps.dwSize = sizeof(ddcaps);

// Now the structure can be used.
.
```

D3DCLIPSTATUS

The **D3DCLIPSTATUS** structure describes the current clip status and extents of the clipping region. This structure was introduced in DirectX 5.0.

```
typedef struct _D3DCLIPSTATUS {
    DWORD dwFlags;
```

```

    DWORD dwStatus;
    float minx, maxx;
    float miny, maxy;
    float minz, maxz;
} D3DCLIPSTATUS, *LPD3DCLIPSTATUS;

```

Members

dwFlags

Flags describing whether this structure describes 2-D extents, 3-D extents, or the clip status. This member can be a combination of the following flags:

D3DCLIPSTATUS_STATUS

This structure describes the current clip status.

D3DCLIPSTATUS_EXTENTS2

This structure describes the current 2-D extents. This flag cannot be combined with D3DCLIPSTATUS_EXTENTS3.

D3DCLIPSTATUS_EXTENTS3

Not currently implemented.

dwStatus

Describes the current clip status. This member can be one or more of the following flags:

Combination and General Flags

D3DSTATUS_CLIPINTERSECTIONALL

Combination of all CLIPINTERSECTION flags.

D3DSTATUS_CLIPUNIONALL

Combination of all CLIPUNION flags.

D3DSTATUS_DEFAULT

Combination of D3DSTATUS_CLIPINTERSECTIONALL and D3DSTATUS_ZNOTVISIBLE flags. This value is the default.

D3DSTATUS_ZNOTVISIBLE

Indicates that the rendered primitive is not visible. This flag is set or cleared by the system when rendering with z-checking enabled (see D3DRENDERSTATE_ZVISIBLE).

Clip Intersection Flags

D3DSTATUS_CLIPINTERSECTIONBACK

Logical **AND** operation on the clip flags for the vertices, compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM

Logical **AND** operation on the clip flags for the vertices, compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT

Logical **AND** operation on the clip flags for the vertices, compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through

D3DSTATUS_CLIPINTERSECTIONGEN5

Logical **AND** operation on the clip flags for application-defined clipping planes.

D3DSTATUS_CLIPINTERSECTIONLEFT

Logical **AND** operation on the clip flags for the vertices, compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT

Logical **AND** operation on the clip flags for the vertices, compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP

Logical **AND** operation on the clip flags for the vertices, compared to the top of the viewing frustum.

Clip Union Flags

D3DSTATUS_CLIPUNIONBACK

Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM

Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT

Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5

Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT

Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT

Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP

Equal to D3DCLIP_TOP.

Basic Clipping Flags

D3DCLIP_BACK

All vertices are clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM

All vertices are clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT

All vertices are clipped by the front plane of the viewing frustum.

D3DCLIP_LEFT

All vertices are clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT

All vertices are clipped by the right plane of the viewing frustum.

D3DCLIP_TOP

All vertices are clipped by the top plane of the viewing frustum.

D3DCLIP_GEN0 through D3DCLIP_GEN5

Application-defined clipping planes.

minx, maxx, miny, maxy, minz, maxx

x, y, and z extents of the current clipping region.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::GetClipStatus, IDirect3DDevice7::SetClipStatus

D3DCOLORVALUE

The **D3DCOLORVALUE** structure describes color values for the **D3DMATERIAL7** structure.

```
typedef struct _D3DCOLORVALUE {
    union {
        D3DVALUE r;
        D3DVALUE dvR;
    };
    union {
        D3DVALUE g;
        D3DVALUE dvG;
    };
    union {
        D3DVALUE b;
        D3DVALUE dvB;
    };
    union {
        D3DVALUE a;
        D3DVALUE dvA;
    };
} D3DCOLORVALUE;
```

Members

dvR, dvG, dvB, and dvA

Values of the **D3DVALUE** type 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 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, which actually remove light from a scene. For more information, see [Colored Lights](#).

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3dtypes.h.

D3DDEVICEDESC7

The **D3DDEVICEDESC7** structure contains a description of the current device. This structure is used to query the current device by such methods as

IDirect3DDevice7::GetCaps.

This structure supersedes the **D3DDEVICEDESC** structure for all device interfaces later than **IDirect3DDevice3**.

```
typedef struct _D3DDeviceDesc7 {
    DWORD        dwDevCaps;
    D3DPRIMCAPS  dpcLineCaps;
    D3DPRIMCAPS  dpcTriCaps;
    DWORD        dwDeviceRenderBitDepth;
    DWORD        dwDeviceZBufferBitDepth;
    DWORD        dwMinTextureWidth, dwMinTextureHeight;
    DWORD        dwMaxTextureWidth, dwMaxTextureHeight;
    DWORD        dwMaxTextureRepeat;
    DWORD        dwMaxTextureAspectRatio;
    DWORD        dwMaxAnisotropy;
    D3DVALUE     dvGuardBandLeft;
    D3DVALUE     dvGuardBandTop;
    D3DVALUE     dvGuardBandRight;
    D3DVALUE     dvGuardBandBottom;
    D3DVALUE     dvExtentsAdjust;
    DWORD        dwStencilCaps;
    DWORD        dwFVFCaps;
    DWORD        dwTextureOpCaps;
    WORD         wMaxTextureBlendStages;
    WORD         wMaxSimultaneousTextures;
    DWORD        dwMaxActiveLights;
```

```

D3DVALUE    dvMaxVertexW;
GUID        deviceGUID;
WORD        wMaxUserClipPlanes;
WORD        wMaxVertexBlendMatrices;
DWORD       dwVertexProcessingCaps;
DWORD       dwReserved1;
DWORD       dwReserved2;
DWORD       dwReserved3;
DWORD       dwReserved4;
} D3DDEVICEDESC7, *LPD3DDEVICEDESC7;

```

Members

dwDevCaps

Flags identifying the capabilities of the device.

D3DDEVCAPS_CANBLTSYSTONONLOCAL

Device supports blits from system-memory textures to nonlocal video-memory textures.

This flag was introduced with DirectX 7.0

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.

This flag was introduced in DirectX 5.0.

D3DDEVCAPS_DRAWPRIMTLVERTEX

Device exports a DrawPrimitive-aware hardware abstraction layer (HAL).

This flag was introduced in DirectX 5.0.

D3DDEVCAPS_EXECUTESYSTEMMEMORY

Device can use execute buffers from system memory.

D3DDEVCAPS_EXECUTEVIDEOMEMORY

Device can use execute buffers from video memory.

D3DDEVCAPS_FLOATTLVERTEX

Device accepts floating point for post-transform vertex data.

D3DDEVCAPS_HWRASTERIZATION

Device has hardware acceleration for scene rasterization.

D3DDEVCAPS_HWTRANSFORMANDLIGHT

Device supports transformation and lighting in hardware.

This flag was introduced with DirectX 7.0.

D3DDEVCAPS_SEPARATETEXTUREMEMORIES

Device uses discrete texture memory pools for each stage. Textures must be assigned to texture stages explicitly at the time of creation by setting the **dwTextureStage** member of the **DDSURFACEDESC2** structure to the appropriate stage identifier.

D3DDEVCAPS_SORTDECREASINGZ

Device needs data sorted for decreasing depth.

D3DDEVCAPS_SORTEXACT

Device needs data sorted exactly.

D3DDEVCAPS_SORTINCREASINGZ

Device needs data sorted for increasing depth.

D3DDEVCAPS_STRIDEDVERTICES

Device supports strided vertex data for transformation and lighting in hardware.

This flag was introduced with DirectX 7.0

D3DDEVCAPS_TEXTURENONLOCALVIDMEM

Device can retrieve textures from nonlocal video (AGP) memory.

This flag was introduced in DirectX 5.0. For more information about AGP memory, see Using Non-local Video Memory Surfaces in the DirectDraw documentation.

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.

dpcLineCaps and dpcTriCaps

D3DPRIMCAPS structures defining the device's support for line-drawing and triangle primitives.

dwDeviceRenderBitDepth

Device's rendering bit depth. This can be one or more of the following DirectDraw bit-depth constants: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

dwDeviceZBufferBitDepth

Bit depth of the device's depth-buffer. This can be one of the following DirectDraw bit-depth constants: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

dwMinTextureWidth, dwMinTextureHeight

Minimum texture width and height for this device.

dwMaxTextureWidth, dwMaxTextureHeight

Maximum texture width and height for this device.

dwMaxTextureRepeat

Full range of the integer bits of the post-normalized texture indices. If the **D3DPTEXTURECAPS_TEXREPEATNOTSCALEDDBYSIZE** 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.

dwMaxTextureAspectRatio

Maximum texture aspect ratio supported by the hardware; this is typically a power of 2.

dwMaxAnisotropy

Maximum valid value for the D3DTSS_MAXANISOTROPY texture-stage state.

dvGuardBandLeft, dvGuardBandTop, dvGuardBandRight, and dvGuardBandBottom

The screen-space coordinates of the guard-band clipping region. Coordinates inside this rectangle but outside the viewport rectangle are automatically clipped.

dwExtentsAdjust

Number of pixels to adjust the extents rectangle outward to accommodate antialiasing kernels.

dwStencilCaps

Flags specifying supported stencil-buffer operations. Stencil operations are assumed to be valid for all three stencil-buffer operation render states (D3DRENDERSTATE_STENCILFAIL, D3DRENDERSTATE_STENCILPASS, and D3DRENDERSTATE_STENCILFAILZFFAIL).

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.

dwFVFCaps

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_TEXCOORDCOUNTMASK

Masks the low **WORD** of **dwFVFCaps**. 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 only blend using the specified number of texture coordinate sets.)

dwTextureOpCaps

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_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_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.

wMaxTextureBlendStages

Maximum number of texture-blending stages supported.

wMaxSimultaneousTextures

Maximum number of textures that can be simultaneously bound to the texture blending stages. See remarks.

dwMaxActiveLights

Maximum number of lights that can be active simultaneously.

dvMaxVertexW

Maximum W-based depth value that the device supports.

deviceGUID

Globally unique identifier (GUID) of this device: IID_IDirect3DHALDevice, IID_IDirect3DTnLHalDevice, or IID_IDirect3DRGBDevice.

wMaxUserClipPlanes

Maximum number of user-defined clipping planes supported. This member can range from 0 through D3DMAXUSERCLIPPLANES.

User-defined clipping planes are manipulated by using the

IDirect3DDevice7::GetClipPlane and **IDirect3DDevice7::SetClipPlane** methods.

wMaxVertexBlendMatrices

Maximum number of matrices that this device can apply when performing multimatrix vertex blending.

dwVertexProcessingCaps

Vertex processing capabilities.

D3DVTXPCAPS_TEXGEN

Device can generate texture coordinates.

D3DVTXPCAPS_MATERIALSOURCE7

Device supports selectable vertex color sources. For more information, see Light Color Types and Sources.

D3DVTXPCAPS_VERTEXFOG

Device supports vertex fog.

D3DVTXPCAPS_DIRECTIONALLIGHTS

Device supports directional lights.

D3DVTXPCAPS_POSITIONALLIGHTS

Device supports positional lights (including point lights and spotlights).

D3DVTXPCAPS_NONLOCALVIEWER

Device supports orthogonal specular highlights, enabled by setting the **D3DRENDERSTATE_LOCALVIEWER** render state to **FALSE**.

dwReserved1 through dwReserved4

Reserved.

Remarks

The **wMaxTextureBlendStages** and **wMaxSimultaneousTextures** members might seem very similar, but they contain different information. The **wMaxTextureBlendStages** member contains the total number of texture-blending stages supported by the current device, and the **wMaxSimultaneousTextures** member describes how many of those stages can have textures bound to them by using the **IDirect3DDevice7::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 **IDirect3DDevice7**) does not support execute buffers.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3dcaps.h.

See Also

D3DCOLORMODEL, **D3DLIGHTINGCAPS**, **D3DPRIMCAPS**

D3DDEVINFO_TEXTUREMANAGER

The **D3DDEVINFO_TEXTUREMANAGER** structure contains information about the current state of the texture manager. This structure is used with the **IDirect3DDevice7::GetInfo** method.

```
typedef struct _D3DDEVINFO_TEXTUREMANAGER {
    BOOL    bThrashing;
    DWORD   dwNumEvicts;
    DWORD   dwNumVidCreates;
    DWORD   dwNumTexturesUsed;
    DWORD   dwNumUsedTexInVid;
    DWORD   dwWorkingSet;
    DWORD   dwWorkingSetBytes;
    DWORD   dwTotalManaged;
    DWORD   dwTotalBytes;
    DWORD   dwLastPri;
} D3DDEVINFO_TEXTUREMANAGER,
*LPD3DDEVINFO_TEXTUREMANAGER;
```

Members

bThrashing

Thrashing status. TRUE if thrashing occurred during the last frame, or FALSE otherwise.

dwNumEvicts

Number of textures that were removed during the last frame.

dwNumVidCreates

Number of textures that were created in video memory during the last frame.

dwNumTexturesUsed

Total number of textures used during the last frame.

dwNumUsedTexInVid

Number of video memory textures that were used during the last frame.

dwWorkingSet

Number of textures currently resident in video memory.

dwWorkingSetBytes

Number of bytes currently allocated by textures resident in video memory.

dwTotalManaged

Total number of managed textures.

dwTotalBytes

Total number of bytes allocated for managed textures.

dwLastPri

Priority of last evicted texture.

Remarks

Use the values in **dwNumUsedTexInVid** and **dwNumTexturesUsed** to determine the percentage of textures used during the last frame that were resident in video memory (local or nonlocal). The following code fragment shows how to make this computation:

```
// For this example, the d3dtmInfo variable is a D3DDEVINFO_TEXTUREMANAGER
// structure that has been filled by a call to IDirect3DDevice7::GetInfo.

DWORD dwHitRate;

// Retrieve the percentage of textures used that were in video memory.
dwHitRate = (d3dtmInfo.dwNumUsedTexInVid * 100) / d3dtmInfo.dwNumTexturesUsed;
```

See Also

IDirect3DDevice7::GetInfo

D3DDEVINFO_TEXTUREING

The **D3DDEVINFO_TEXTUREING** structure contains information about the texturing activity of the application. This structure is used with the **IDirect3DDevice7::GetInfo** method.

```
typedef struct _D3DDEVINFO_TEXTUREING {
    DWORD  dwNumLoads;
    DWORD  dwApproxBytesLoaded;
    DWORD  dwNumPreLoads;
    DWORD  dwNumSet;
    DWORD  dwNumCreates;
    DWORD  dwNumDestroys;
    DWORD  dwNumSetPriorities;
    DWORD  dwNumSetLODs;
    DWORD  dwNumLocks;
    DWORD  dwNumGetDCs;
} D3DDEVINFO_TEXTUREING, *LPD3DDEVINFO_TEXTUREING;
```

Members

dwNumLoads

Number of times a texture has been loaded by calling the **IDirect3DDevice7::Load** method.

dwApproxBytesLoaded

Approximate number of bytes loaded by calls to the **IDirect3DDevice7::Load** method.

dwNumPreLoads

Number of times managed textures have been explicitly loaded by calling the **IDirect3DDevice7::PreLoad** method.

dwNumSet

Number of times textures have been set to texture-blending stages by calling the **IDirect3DDevice7::SetTexture** method.

dwNumCreates

Number of texture surfaces created by the application.

dwNumDestroys

Number of textures destroyed (released) by the application.

dwNumSetPriorities

Number of times texture-management priority has been set by calling the **IDirectDrawSurface7::SetPriority** method.

dwNumSetLODs

Number of times the maximum mipmap level of detail has been set by calling the **IDirectDrawSurface7::SetLOD** method.

dwNumLocks

Number of times a texture surface has been locked by calling the **IDirectDrawSurface7::Lock** method.

dwNumGetDCs

Number of times a device context for a texture surface has been retrieved by calling the **IDirectDrawSurface7::GetDC** method.

See Also

IDirect3DDevice7::GetInfo

D3DDP_PTRSTRIDE

The **D3DDP_PTRSTRIDE** structure contains the address of an array of flexible vertex format components and the stride to the next element in the array. This structure is contained in the **D3DDRAWPRIMITIVESTRIDEEDDATA** structure.

```
typedef struct _D3DDP_PTRSTRIDE {  
    LPVOID lpvData;  
    DWORD dwStride;  
} D3DDP_PTRSTRIDE;
```

Members**lpvData**

Address of an array of data.

dwStride

Memory stride between elements in the array.

Remarks

This structure can be used with a composite vertex format (like the **D3DLVERTEX** structure) or a distinct array of vertex components. In a composite vertex format, **lpvData** points to a particular component, and **dwStride** is the stride, in bytes, of the composite format. In an array that contains only one vertex component, **dwStride** is the stride of each element in the array.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DDRAWPRIMITIVESTRIDEDDATA, Strided Vertex Format

D3DDRAWPRIMITIVESTRIDEDDATA

The **D3DDRAWPRIMITIVESTRIDEDDATA** structure contains flexible vertex format components.

```
typedef struct D3DDRAWPRIMITIVESTRIDEDDATA {
    D3DDP_PTRSTRIDE position;
    D3DDP_PTRSTRIDE normal;
    D3DDP_PTRSTRIDE diffuse;
    D3DDP_PTRSTRIDE specular;
    D3DDP_PTRSTRIDE textureCoords[D3DDP_MAXTEXCOORD];
} D3DDRAWPRIMITIVESTRIDEDDATA,
*LPD3DDRAWPRIMITIVESTRIDEDDATA;
```

Members

position and normal

D3DDP_PTRSTRIDE structures that point to arrays of position and normal vectors for a collection of vertices (each vector is a three-element array of float values).

diffuse and specular

D3DDP_PTRSTRIDE structures that point to diffuse and specular color information for a collection of vertices. Each color component is an 8-8-8-8 RGBA value.

textureCoords

An eight-element array of **D3DDP_PTRSTRIDE** structures. Each element in the array is an array of texture coordinates for the collection of vertices. Your application determines which array of texture coordinates is used for a given texture stage by calling the **IDirect3DDevice7::SetTextureStageState** method with the **D3DTSS_TEXCOORDINDEX** stage state value.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DDP_PTRSTRIDE, Strided Vertex Format, Vertex Formats

D3DLIGHT7

The **D3DLIGHT7** structure defines a set of lighting properties. This structure is used with the **IDirect3DDevice7::GetLight** and **IDirect3DDevice7::SetLight** methods.

```
typedef struct _D3DLIGHT7 {
    D3DLIGHTTYPE    dltType;
    D3DCOLORVALUE   dcvDiffuse;
    D3DCOLORVALUE   dcvSpecular;
    D3DCOLORVALUE   dcvAmbient;
    D3DVECTOR        dvPosition;
    D3DVECTOR        dvDirection;
    D3DVALUE         dvRange;
    D3DVALUE         dvFalloff;
    D3DVALUE         dvAttenuation0;
    D3DVALUE         dvAttenuation1;
    D3DVALUE         dvAttenuation2;
    D3DVALUE         dvTheta;
    D3DVALUE         dvPhi;
} D3DLIGHT7, *LPD3DLIGHT7;
```

Members

dltType

Type of the light source. This value is one of the members of the **D3DLIGHTTYPE** enumerated type.

dcvDiffuse

Diffuse color emitted by the light. This member is a **D3DCOLORVALUE** structure.

dcvSpecular

Specular color emitted by the light. This member is a **D3DCOLORVALUE** structure.

dcvAmbient

Ambient color emitted by the light. This member is a **D3DCOLORVALUE** structure.

dvPosition

Position of the light in world space. This member has no meaning for directional lights and is ignored in that case.

dvDirection

Direction that the light is pointing in world space. This member only has meaning for directional and spotlights. This vector need not be normalized, but it should have a nonzero length.

dvRange

Distance beyond which the light has no effect. The maximum allowable value for this member is **D3DLIGHT_RANGE_MAX**, which is defined as the square root of **FLT_MAX**. This member does not affect directional lights.

dvFalloff

Decrease in illumination between a spotlight's inner cone (the angle specified by **dvTheta**) and the outer edge of the outer cone (the angle specified by **dvPhi**). This feature was implemented for DirectX 5.0. For details on how **dvFalloff** values affect a spotlight, see Spotlight Falloff Model.

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.

dvAttenuation0, dvAttenuation1, and dvAttenuation2

Values specifying how the light intensity changes over distance. (Attenuation does not affect directional lights.) These members represent a light's constant, linear, and quadratic attenuation factors. For information about how these attenuation values affect lighting in a scene, see Light Attenuation Over Distance. Valid values for these members range from 0.0 to infinity.

dvTheta

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 **dvPhi**.

dvPhi

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 π .

Remarks

This structure, introduced with DirectX 7.0, supersedes the **D3DLIGHT2** and **D3DLIGHT** structures used with previous versions of DirectX.

This structure differs from the **D3DLIGHT2** structure in that it provides members for independent diffuse, specular, and ambient colors of a light. Each color value is used in the lighting module of Direct3D Immediate Mode to determine the lighting in a scene.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::GetLight, **IDirect3DDevice7::SetLight**, **D3DLIGHTTYPE**, Lighting and Materials

D3DLIGHTINGCAPS

The **D3DLIGHTINGCAPS** structure describes the lighting capabilities of a device. This structure is a member of the **D3DDEVICEDESC7** structure.

```
typedef struct _D3DLIGHTINGCAPS {
    DWORD dwSize;
    DWORD dwCaps;
    DWORD dwLightingModel;
    DWORD dwNumLights;
} D3DLIGHTINGCAPS, *LPD3DLIGHTINGCAPS;
```

Members

dwSize

Size, in bytes, of this structure. This member must be initialized before the structure is used.

dwCaps

Flags describing the capabilities of the lighting module. The following flags are defined:

D3DLIGHTCAPS_DIRECTIONAL

Supports directional lights.

D3DLIGHTCAPS_GLSPOT

Not used.

D3DLIGHTCAPS_POINT

Supports point lights.

D3DLIGHTCAPS_SPOT

Supports spotlights.

dwLightingModel

Flags defining whether the lighting model is RGB or monochrome. The following flags are defined:

D3DLIGHTINGMODEL_MONO

Monochromatic lighting model.

D3DLIGHTINGMODEL_RGB

RGB lighting model.

dwNumLights

Number of lights that can be handled.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dcaps.h.

D3DLINEPATTERN

The **D3DLINEPATTERN** structure describes a line pattern. These values are used by the D3DRENDERSTATE_LINEPATTERN render state in the **D3DRENDERSTATETYPE** enumerated type.

```
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

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

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dypes.h.

D3DLVERTEX

The **D3DLVERTEX** structure defines an untransformed lit vertex (model coordinates with color). An application should use this structure when vertex transformations are to be handled by Direct3D. This structure contains only data and a color that would be filled by software lighting.

```
typedef struct _D3DLVERTEX {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
    DWORD    dwReserved;
    union {
        D3DCOLOR color;
        D3DCOLOR dcColor;
    };
    union {
        D3DCOLOR specular;
        D3DCOLOR dcSpecular;
    };
};
```

```

union {
    D3DVALUE tu;
    D3DVALUE dvTU;
};
union {
    D3DVALUE tv;
    D3DVALUE dvTV;
};
} D3DLVERTEX, *LPD3DLVERTEX;

```

Members

dvX, dvY, and dvZ

Values of the **D3DVALUE** type specifying the model coordinates of the vertex.

dwReserved

Reserved; must be 0.

dcColor and dcSpecular

Values of the **D3DCOLOR** type specifying the color and specular component of the vertex.

dvTU and dvTV

Values of the **D3DVALUE** type specifying the texture coordinates of the vertex.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DTLVERTEX, D3DVERTEX

D3DMATERIAL7

The **D3DMATERIAL7** structure specifies material properties in calls to the **IDirect3DDevice7::GetMaterial** and **IDirect3DDevice7::SetMaterial** methods.

```

typedef struct _D3DMATERIAL7 {
    union {
        D3DCOLORVALUE  diffuse;
        D3DCOLORVALUE  dcvDiffuse;
    };
};

```

```

union {
    D3DCOLORVALUE  ambient;
    D3DCOLORVALUE  dcvAmbient;
};
union {
    D3DCOLORVALUE  specular;
    D3DCOLORVALUE  dcvSpecular;
};
union {
    D3DCOLORVALUE  emissive;
    D3DCOLORVALUE  dcvEmissive;
};
union {
    D3DVALUE        power;
    D3DVALUE        dvPower;
};
} D3DMATERIAL7, *LPD3DMATERIAL7;

```

Members

dcvDiffuse, dcvAmbient, dcvSpecular, and dcvEmissive

Values specifying the diffuse color, ambient color, specular color, and emissive color of the material, respectively. These values are **D3DCOLORVALUE** structures.

dvPower

Value of the **D3DVALUE** type 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.

Remarks

This structure, introduced with DirectX 7.0, supersedes the **D3DMATERIAL** structure used with previous versions of DirectX.

This structure, unlike the legacy **D3DMATERIAL** structure, does not support texture handles (for textured viewport background) or ramp sizes for the legacy Ramp software device.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::GetMaterial, **IDirect3DDevice7::SetMaterial**, Lighting and Materials

D3DMATRIX

The **D3DMATRIX** structure describes a matrix for such methods as **IDirect3DDevice7::MultiplyTransform**, **IDirect3DDevice7::GetTransform**, and **IDirect3DDevice7::SetTransform**.

C++ programmers can use an extended version of this structure that includes a parentheses ("()") operator. For more information, see **D3DMATRIX (D3D_OVERLOADS)**

```
typedef struct _D3DMATRIX {
    D3DVALUE _11, _12, _13, _14;
    D3DVALUE _21, _22, _23, _24;
    D3DVALUE _31, _32, _33, _34;
    D3DVALUE _41, _42, _43, _44;
} D3DMATRIX, *LPD3DMATRIX;
```

Remarks

In 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

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::MultiplyTransform, **IDirect3DDevice7::GetTransform**, **IDirect3DDevice7::SetTransform**

D3DPRIMCAPS

The **D3DPRIMCAPS** structure defines the capabilities for each primitive type. This structure is used when creating a device and when querying the capabilities of a

device. This structure defines several members of the **D3DDEVICEDESC7** structure.

```
typedef struct _D3DPrimCaps {
    DWORD dwSize;           //Size of structure
    DWORD dwMiscCaps;       //Miscellaneous capabilities
    DWORD dwRasterCaps;     //Raster capabilities
    DWORD dwZCmpCaps;       //Z-comparison capabilities
    DWORD dwSrcBlendCaps;    //Source-blending capabilities
    DWORD dwDestBlendCaps;  //Destination-blending capabilities
    DWORD dwAlphaCmpCaps;   //Alpha-test-comparison capabilities
    DWORD dwShadeCaps;       //Shading capabilities
    DWORD dwTextureCaps;    //Texture capabilities
    DWORD dwTextureFilterCaps; //Texture-filtering capabilities
    DWORD dwTextureBlendCaps; //Texture-blending capabilities
    DWORD dwTextureAddressCaps; //Texture-addressing capabilities
    DWORD dwStippleWidth;    //Stipple width
    DWORD dwStippleHeight;   //Stipple height
} D3DPRIMCAPS, *LPD3DPRIMCAPS;
```

Members

dwSize

Size, in bytes, of this structure. This member must be initialized before the structure is used.

dwMiscCaps

General capabilities for this primitive. This member can be one or more of the following:

D3DPMISCCAPS_CONFORMANT

The device conforms to the OpenGL standard.

D3DPMISCCAPS_CULLCCW

The driver supports counterclockwise culling through the **D3DRENDERSTATE_CULLMODE** state. (This applies only to triangle primitives.) This corresponds to the **D3DCULL_CCW** member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_CULLCW

The driver supports clockwise triangle culling through the **D3DRENDERSTATE_CULLMODE** state. (This applies only to triangle primitives.) This 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_MASKPLANES

The device can perform a bitmask of color planes.

D3DPMISCCAPS_MASKZ

The device can enable and disable modification of the depth buffer on pixel operations.

dwRasterCaps

Information on raster-drawing capabilities. This member can be one or more of the following:

D3DPRASTERCAPS_ANISOTROPY

The device supports anisotropic filtering. For more information, see **D3DRENDERSTATE_ANISOTROPY** in the **D3DRENDERSTATETYPE** structure.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_ANTIALIASEDGES

The device can antialias lines forming the convex outline of objects. For more information, see **D3DRENDERSTATE_EDGEANTIALIAS** in the **D3DRENDERSTATETYPE** structure.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT

The device supports antialiasing that is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur. For more information, see the **D3DANTIALIASMODE** enumerated type.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT

The device supports antialiasing that is not dependent on the sort order of the polygons. For more information, see the **D3DANTIALIASMODE** enumerated type.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_DITHER

The device can dither to improve color resolution.

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.

This flag was introduced in DirectX 5.0.

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_FOGVERTEX

The device calculates the fog value during the lighting operation, places the value into the alpha component of the **D3DCOLOR** value given for the **specular** member of the **D3DTLVERTEX** structure and interpolates the fog value during rasterization.

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_PAT

The driver can perform patterned drawing (lines or fills with **D3DRENDERSTATE_LINEPATTERN** or one of the **D3DRENDERSTATE_STIPPLEPATTERN** render states) for the primitive being queried.

D3DPRASTERCAPS_ROP2

The device can support raster operations other than **R2_COPYPEN**.

D3DPRASTERCAPS_STIPPLE

The device can stipple polygons to simulate translucency.

D3DPRASTERCAPS_SUBPIXEL

The device performs subpixel placement of z, color, and texture data, rather than working with the nearest integer pixel coordinate. This helps avoid bleed-through due to z imprecision and jitter of color and texture values for pixels. There is no corresponding state that can be enabled and disabled; the device either performs subpixel placement, or it does not. This bit is present only so that the you can better determine what the rendering quality will be.

D3DPRASTERCAPS_SUBPIXELX

The device is subpixel-accurate along the x-axis only and is clamped to an integer y-axis scan line. For information about subpixel accuracy, see **D3DPRASTERCAPS_SUBPIXEL**.

D3DPRASTERCAPS_TRANSLUCENTSORTINDEPENDENT

The device supports translucency that is not dependent on the sort order of the polygons. For more information, see **D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT**.

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_XOR

The device can support **XOR** operations. If this flag is not set but **D3DPRIM_RASTER_ROP2** is set, **XOR** operations must still be supported.

D3DPRASERCAPS_ZBIAS

The device supports z-bias values. These are integer values assigned to polygons that allow physically coplanar polygons to appear separate. For more information, see D3DRENDERSTATE_ZBIAS in the **D3DRENDERSTATETYPE** enumerated type.

This flag was introduced in DirectX 5.0.

D3DPRASERCAPS_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 hidden-surface removal is hardware-dependent and is transparent to the application.

Z-bufferless HSR is performed if no depth-buffer surface is attached to the rendering-target surface and the depth-buffer comparison test is enabled (that is, when the state value associated with the D3DRENDERSTATE_ZENABLE enumeration constant is set to TRUE).

D3DPRASERCAPS_ZFOG

The device supports z-based fog.

This flag was introduced in DirectX 7.0.

D3DPRASERCAPS_ZTEST

The device can perform z-test operations. This effectively renders a primitive and indicates whether any z pixels have been rendered.

dwZCmpCaps

Z-buffer comparison capabilities. This member can be one or more of the following:

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.

dwSrcBlendCaps

Source-blending capabilities. This member can be one or more of the following. (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 DirectX 6.0 and later. 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_INVSRCOLOR

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 $(f, f, f, 1)$; $f = \min(A_s, 1-A_d)$.

D3DPBLENDCAPS_SRCCOLOR

Blend factor is (R_s, G_s, B_s, A_s) .

D3DPBLENDCAPS_ZERO

Blend factor is $(0, 0, 0, 0)$.

dwDestBlendCaps

Destination-blending capabilities. This member can be the same capabilities that are defined for the **dwSrcBlendCaps** member.

dwAlphaCmpCaps

Alpha-test comparison capabilities. This member can include the same capability flags defined for the **dwZCmpCaps** 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.

dwShadeCaps

Shading operations capabilities. It is assumed, in general, that if a device supports a given command (such as D3DOP_TRIANGLE) 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 and Phong shading and whether alpha color components are supported for each of the three color-generation modes. When alpha components are not supported in a given mode, the alpha value of colors generated in that mode is implicitly 255. This is the maximum possible alpha (that is, the alpha component is at full intensity). Phong shading is not currently supported.

With the monochromatic shading modes, the blue channel of the specular component is interpreted as a white intensity. (This is controlled by the D3DRENDERSTATE_MONOENABLE render state.)

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. These are modified by the shading mode, color model, and whether the alpha component of a color is blended or stippled. For more information, see 3-D Primitives.

This member can be one or more of the following:

D3DPSHADECAPS_ALPHAFLATBLEND

D3DPSHADECAPS_ALPHAFLATSTIPPLED

Device can support an alpha component for flat blended and stippled transparency, respectively (the D3DSHADE_FLAT state for the **D3DSHADEMODE** enumerated type). In these modes, the alpha color component for a primitive is provided as part of the color for the first vertex of the primitive.

D3DPSHADECAPS_ALPHAGOURAUBLEND

D3DPSHADECAPS_ALPHAGOURAUDSTIPPLED

Device can support an alpha component for Gouraud-blended and stippled transparency, respectively (the D3DSHADE_GOURAUD state for the **D3DSHADEMODE** enumerated type). In these modes, the alpha color component of a primitive is provided at vertices and interpolated across a face along with the other color components.

D3DPSHADECAPS_ALPHAPHONGBLEND

D3DPSHADECAPS_ALPHAPHONGSTIPPLED

Device can support an alpha component for Phong-blended and stippled transparency, respectively (the D3DSHADE_PHONG state for the **D3DSHADEMODE** enumerated type). In these modes, vertex parameters are reevaluated on a per-pixel basis, applying lighting effects for the red, green, and blue color components. Phong shading is not currently supported.

D3DPSHADECAPS_COLORFLATMONO

D3DPSHADECAPS_COLORFLATRGB

Device can support colored flat shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, the color component for a primitive is provided as part of the color for the first vertex of the primitive. In the monochromatic lighting model, only the blue component

of the color is interpolated; in the RGB lighting model, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORGOURAUDMONO

D3DPSHADECAPS_COLORGOURAUDRGB

Device can support colored Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, the color component for a primitive is provided at vertices and interpolated across a face along with the other color components. In the monochromatic lighting model, only the blue component of the color is interpolated; in the RGB lighting model, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORPHONGMONO

D3DPSHADECAPS_COLORPHONGRGB

Device can support colored Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, vertex parameters are reevaluated on a per-pixel basis. Lighting effects are applied for the red, green, and blue color components in the RGB model, and for the blue component only for the monochromatic model. Phong shading is not currently supported.

D3DPSHADECAPS_FOGFLAT

D3DPSHADECAPS_FOGGOURAUD

D3DPSHADECAPS_FOGPHONG

Device can support fog in the flat, Gouraud, and Phong shading modes, respectively. Phong shading is not currently supported.

D3DPSHADECAPS_SPECULARFLATMONO

D3DPSHADECAPS_SPECULARFLATRGB

Device can support specular highlights in flat shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARGOURAUDMONO

D3DPSHADECAPS_SPECULARGOURAUDRGB

Device can support specular highlights in Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARPHONGMONO

D3DPSHADECAPS_SPECULARPHONGRGB

Device can support specular highlights in Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. Phong shading is not currently supported.

dwTextureCaps

Miscellaneous texture-mapping capabilities. This member can be one or more of the following:

D3DPTURECAPS_ALPHA

Supports RGBA textures in the **D3DTBLEND_DECAL** and **D3DTBLEND_MODULATE** texture filtering modes. If this capability is not set, only RGB textures are supported in those modes. Regardless of the setting of this flag, alpha must always be supported in **D3DTBLEND_DECALMASK**, **D3DTBLEND_DECALALPHA**, and

D3DTBLEND_MODULATEALPHA filtering modes whenever those filtering modes are available.

D3DPTEXTURECAPS_ALPHAPALETTE

Supports palettized texture surfaces whose palettes contain alpha information (see DDPCAPS_ALPHA in the **DDCAPS** structure).

D3DPTEXTURECAPS_BORDER

Superseded by D3DPTADDRESSCAPS_BORDER.

D3DPTEXTURECAPS_COLORKEYBLEND

The device supports alpha-blended colorkeying through the use of the D3DRENDERSTATE_COLORKEYBLENDENABLE render state.

D3DPTEXTURECAPS_CUBEMAP

Supports cubic environment mapping. This capability flag was introduced with DirectX 7.0

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 D3DTEXTADDRESS_CLAMP.
- Texture wrapping for the texture stage is disabled (D3DRENDERSTATE_WRAP n set to 0).
- Mipmapping is not in use. (Mipmapped textures must have dimensions that are powers of 2.)

D3DPTEXTURECAPS_PERSPECTIVE

Perspective correction is supported.

D3DPTEXTURECAPS_POW2

All nonmipmapped textures must have widths and heights specified as powers of 2. (Mipmapped textures must always have dimensions that are powers of 2.)

D3DPTEXTURECAPS_PROJECTED

Supports the D3DTTFF_PROJECTED texture transformation flag. When applied, the device divides transformed texture coordinates by the last texture coordinate.

D3DPTEXTURECAPS_SQUAREONLY

All textures must be square.

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDDBYSIZE

Texture indices are not scaled by the texture size prior to interpolation.

This flag was introduced with DirectX 7.0

D3DPTEXTURECAPS_TRANSPARENCY

Texture transparency is supported. (Only those texels that are not the current transparent color are drawn.)

dwTextureFilterCaps

Texture-filtering capabilities. 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 **IDirect3DDevice7** interface. This member can be any combination of the following general and per-stage texture-filtering flags:

General texture-filtering flags

D3DPTFILTERCAPS_LINEAR

Bilinear filtering. Chooses the texel that has the nearest coordinates, then performs a weighted average with the four surrounding texels to determine the final color. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, both must be supported.

D3DPTFILTERCAPS_LINEARMIPLINEAR

Trilinear interpolation between mipmaps. Performs bilinear filtering on the two nearest mipmaps, then interpolates linearly between the two colors to determine a final color.

D3DPTFILTERCAPS_LINEARMIPNEAREST

Linear interpolation between two point-sample mipmaps. Chooses the nearest texel from the two closest mipmap levels, then performs linear interpolation between them.

D3DPTFILTERCAPS_MIPLINEAR

Nearest mipmapping, with bilinear filtering applied to the result. Chooses the texel from the appropriate mipmap that has the nearest coordinates, then performs a weighted average with the four surrounding texels to determine the final color.

D3DPTFILTERCAPS_MIPNEAREST

Nearest mipmapping. Chooses the texel from the appropriate mipmap with coordinates nearest to the desired pixel value.

D3DPTFILTERCAPS_NEAREST

Point sampling. The texel with coordinates nearest to the desired pixel value is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, both must be supported.

Per-stage texture filtering flags

D3DPTFILTERCAPS_MAGFAFLATCUBIC

The device supports per-stage flat cubic filtering for magnifying textures. The flat cubic magnification filter is represented by the D3DTFG_FLATCUBIC member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MAGFANISOTROPIC

The device supports per-stage anisotropic filtering for magnifying textures. The anisotropic magnification filter is represented by the D3DTFG_ANISOTROPIC member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC

The device supports the per-stage Gaussian cubic filtering for magnifying textures. The Gaussian cubic magnification filter is represented by the **D3DTFG_GAUSSIANCUBIC** member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MAGFLINEAR

The device supports per-stage bilinear interpolation filtering for magnifying textures. The bilinear interpolation magnification filter is represented by the **D3DTFG_LINEAR** member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MAGFPOINT

The device supports per-stage point-sample filtering for magnifying textures. The point-sample magnification filter is represented by the **D3DTFG_POINT** member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MINFANISOTROPIC

The device supports per-stage anisotropic filtering for minifying textures. The anisotropic minification filter is represented by the **D3DTFN_ANISOTROPIC** member of the **D3DTEXTUREMINFILTER** enumerated type.

D3DPTFILTERCAPS_MINFLINEAR

The device supports per-stage bilinear interpolation filtering for minifying textures. The bilinear minification filter is represented by the **D3DTFN_LINEAR** member of the **D3DTEXTUREMINFILTER** enumerated type.

D3DPTFILTERCAPS_MINFPOINT

The device supports per-stage point-sample filtering for minifying textures. The point-sample minification filter is represented by the **D3DTFN_POINT** member of the **D3DTEXTUREMINFILTER** enumerated type.

D3DPTFILTERCAPS_MIPFLINEAR

The device supports per-stage trilinear interpolation filtering for mipmaps. The trilinear interpolation mipmapping filter is represented by the **D3DTFP_LINEAR** member of the **D3DTEXTUREMIPFILTER** enumerated type.

D3DPTFILTERCAPS_MIPFPOINT

The device supports per-stage point-sample filtering for mipmaps. The point-sample mipmapping filter is represented by the **D3DTFP_POINT** member of the **D3DTEXTUREMIPFILTER** enumerated type.

dwTextureBlendCaps

Not used.

In previous releases of DirectX, this member contained information about the driver's capability for applying texture-blending through the now-obsolete **D3DRENDERSTATE_TEXTUREMAPBLEND** render state. Information about the texture-blending capabilities for a DirectX 7.0 device is exposed in the **dwTextureOpCaps** member of the **D3DDEVICEDESC7** structure.

dwTextureAddressCaps

Texture-addressing capabilities. This member can be one or more of the following:

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.

This flag was introduced in DirectX 5.0.

D3DPTADDRESSCAPS_MIRROR

Device can mirror textures to addresses.

D3DPTADDRESSCAPS_WRAP

Device can wrap textures to addresses.

dwStippleWidth and dwStippleHeight

Maximum width and height of the supported stipple (up to 32×32).

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dcaps.h.

D3DRECT

The **D3DRECT** structure is a rectangle definition.

```
typedef struct _D3DRECT {
    union {
        LONG x1;
        LONG IX1;
    };
    union {
        LONG y1;
        LONG IY1;
    };
    union {
        LONG x2;
        LONG IX2;
    };
    union {
        LONG y2;
```

```
    LONG IY2;  
};  
} D3DRECT, *LPD3DRECT;
```

Members

IX1 and IY1

Coordinates of the upper-left corner of the rectangle.

IX2 and IY2

Coordinates of the lower-right corner of the rectangle.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::Clear

D3DTLVERTEX

The **D3DTLVERTEX** structure defines a transformed and lit vertex (screen coordinates with color).

```
typedef struct _D3DTLVERTEX {  
    union {  
        D3DVALUE sx;  
        D3DVALUE dvSX;  
    };  
    union {  
        D3DVALUE sy;  
        D3DVALUE dvSY;  
    };  
    union {  
        D3DVALUE sz;  
        D3DVALUE dvSZ;  
    };  
    union {  
        D3DVALUE rhw;  
        D3DVALUE dvRHW;  
    };  
};
```

```

union {
    D3DCOLOR color;
    D3DCOLOR dcColor;
};
union {
    D3DCOLOR specular;
    D3DCOLOR dcSpecular;
};
union {
    D3DVALUE tu;
    D3DVALUE dvTU;
};
union {
    D3DVALUE tv;
    D3DVALUE dvTV;
};
} D3DTLVERTEX, *LPD3DTLVERTEX;

```

Parameters

dvSX, dvSY, and dvSZ

Values of the **D3DVALUE** type describing a vertex in screen coordinates. The largest allowable value for **dvSZ** is 0.99999 if you want the vertex to be within the range of z-values that are displayed.

dvRHW

Value of the **D3DVALUE** type 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.

dcColor and dcSpecular

Values of the **D3DCOLOR** type describing the color and specular component of the vertex.

dvTU and dvTV

Values of the **D3DVALUE** type describing the texture coordinates of the vertex.

Remarks

Direct3D uses the current viewport parameters (the **dwX**, **dwY**, **dwWidth**, and **dwHeight** members of the **D3DVIEWPORT7** structure) to clip **D3DTLVERTEX** vertices. The system always clips z-coordinates to [0, 1]. To prevent the system from clipping these vertices, use the **D3DDP_DONOTCLIP** flag when calling rendering methods.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DLVERTEX, **D3DVERTEX**

D3DVECTOR

The **D3DVECTOR** structure defines a vector for many Direct3D and Direct3DRM methods and structures.

```
typedef struct _D3DVECTOR {  
    union {  
        D3DVALUE x;  
        D3DVALUE dvX;  
    };  
    union {  
        D3DVALUE y;  
        D3DVALUE dvY;  
    };  
    union {  
        D3DVALUE z;  
        D3DVALUE dvZ;  
    };  
} D3DVECTOR, *LPD3DVECTOR;
```

Members

dvX, **dvY**, and **dvZ**

Values of the **D3DVALUE** type describing the vector.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

D3DVERTEX

The **D3DVERTEX** structure defines an untransformed and unlit vertex (model coordinates with normal direction vector).

```
typedef struct _D3DVERTEX {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
    union {
        D3DVALUE nx;
        D3DVALUE dvNX;
    };
    union {
        D3DVALUE ny;
        D3DVALUE dvNY;
    };
    union {
        D3DVALUE nz;
        D3DVALUE dvNZ;
    };
    union {
        D3DVALUE tu;
        D3DVALUE dvTU;
    };
    union {
        D3DVALUE tv;
        D3DVALUE dvTV;
    };
} D3DVERTEX, *LPD3DVERTEX;
```

Members

dvX, dvY, and dvZ

Values of the **D3DVALUE** type describing the homogeneous coordinates of the vertex.

dvNX, dvNY, and dvNZ

Values of the **D3DVALUE** type describing the normal coordinates of the vertex.

dvTU and dvTV

Values of the **D3DVALUE** type describing the texture coordinates of the vertex.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DLVERTEX, D3DTLVERTEX, D3DVALUE

D3DVERTEXBUFFERDESC

The **D3DVERTEXBUFFERDESC** structure describes the properties of a vertex buffer object. This structure is used with the **IDirect3D7::CreateVertexBuffer** and **IDirect3DVertexBuffer7::GetVertexBufferDesc** methods.

```
typedef struct _D3DVERTEXBUFFERDESC {
    DWORD dwSize;
    DWORD dwCaps;
    DWORD dwFVF;
    DWORD dwNumVertices;
} D3DVERTEXBUFFERDESC, *LPD3DVERTEXBUFFERDESC;
```

Members

dwSize

Size of this structure, in bytes. This member must be initialized before the structure is used.

dwCaps

Capability flags that describe the vertex buffer and identify whether the vertex buffer can contain optimized vertex data. This parameter can be any combination of the following flags:

(none)

The vertex buffer should be created in whatever memory the driver chooses, to allow efficient read operations.

D3DVBCAPS_DONOTCLIP

The vertex buffer cannot contain clipping information.

D3DVBCAPS_OPTIMIZED

The vertex buffer can contain optimized vertex data. (This flag is not used when creating a new vertex buffer.)

D3DVBCAPS_SYSTEMMEMORY

The vertex buffer should be created in system memory. Use this capability for vertex buffers to be rendered by using software devices (MMX and RGB devices).

D3DVBCAPS_WRITEONLY

Informs the system that the application only writes 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.

dwFVF

A combination of flexible vertex format flags that describes the vertex format of the vertices in this buffer.

dwNumVertices

The maximum number of vertices that this vertex buffer can contain. The maximum number of vertices allowed is D3DMAXNUMVERTICES (0xFFFF).

Remarks

Software devices—MMX and RGB devices—cannot render from a video memory (local or nonlocal) vertex buffer. If you want to render a vertex buffer, using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dtypes.h.

See Also

Vertex Buffer Descriptions, Vertex Buffers

D3DVIEWPORT7

The **D3DVIEWPORT7** structure defines the window dimensions of a render target surface onto which a 3-D volume projects. This structure is used with the **IDirect3DDevice7::GetViewport** and **IDirect3DDevice7::SetViewport** methods.

```
typedef struct _D3DVIEWPORT7{
```

```
DWORD    dwX;  
DWORD    dwY;  
DWORD    dwWidth;  
DWORD    dwHeight;  
D3DVALUE  dvMinZ;  
D3DVALUE  dvMaxZ;  
} D3DVIEWPORT7, *LPD3DVIEWPORT7
```

Members

dwX and **dwY**

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.

dwWidth and **dwHeight**

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.

dvMinZ and **dvMaxZ**

Values of the **D3DVALUE** type describing the range of depth values into which a scene is to be rendered. Most applications set these values to 0.0 and 1.0, respectively. The interpretation of the values in these members is completely different from the corresponding members in the legacy **D3DVIEWPORT2** structure. Clipping in DirectX 7.0 is performed after applying the projection matrix. For more information, see *Clipping Volumes and The Projection Transformation*.

Remarks

This structure, introduced in DirectX 7.0, supersedes the **D3DVIEWPORT2** and **D3DVIEWPORT** structures used with previous versions of DirectX.

The **dwX**, **dwY**, **dwWidth**, and **dwHeight** 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 640x480 surface, these members should be 0, 0, 640, and 480, respectively. The **dvMinZ** and **dvMaxZ** 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 **IDirect3DDevice7::SetViewport** method), the driver builds a new transformation matrix.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3dtypes.h.

See Also

D3DVALUE, **IDirect3DDevice7::GetViewport**, **IDirect3DDevice7::SetViewport**, Clipping Volumes, Viewports and Clipping

Obsolete Structures

The following structures are obsolete. In some cases, the functionality provided by obsolete structures is now present within the **IDirect3DDevice7** interface and its related structures. To summarize the relationships:

D3DDEVICEDESC

Superseded by the **D3DDEVICEDESC7** structure used with the **IDirect3DDevice7::GetCaps** method.

D3DHVERTEX

Not supported.

D3DFINDDEVICESEARCH and **D3DFINDDEVICERESULT**

Obsolete.

D3DLIGHT and **D3DLIGHT2**

Superseded by the **D3DLIGHT7** structure used with the **IDirect3DDevice7::GetLight** and **IDirect3DDevice7::SetLight** methods.

D3DLIGHTDATA

Not supported.

D3DLIGHTINGELEMENT

Not supported.

D3DMATERIAL

Superseded by the **D3DMATERIAL7** structure used with the **IDirect3DDevice7::GetMaterial** and **IDirect3DDevice7::SetMaterial** methods.

D3DSTATE

Not supported.

D3DSTATS

Not supported.

D3DTRANSFORMCAPS

Not supported.

D3DTRANSFORMDATA

Not supported.

D3DVIEWPORT and D3DVIEWPORT2

Superseded by the **D3DVIEWPORT7** structure used with the **IDirect3DDevice7::GetViewport** and **IDirect3DDevice7::SetViewport** methods.

Execute-buffer rendering, and the following related structures, is considered obsolete with DirectX 7.0. For information about these structures, see the documentation provided with a previous release of DirectX. Legacy documentation is provided with this SDK and is available for download from <http://www.microsoft.com/directx>.

- **D3DBRANCH**
- **D3DEXECUTEBUFFERDESC**
- **D3DEXECUTEDATA**
- **D3DINSTRUCTION**
- **D3DLINE**
- **D3DMATRIXLOAD**
- **D3DMATRIXMULTIPLY**
- **D3DPICKRECORD**
- **D3DPOINT**
- **D3DPROCESSVERTICES**
- **D3DSPAN**
- **D3DSTATUS**
- **D3DTEXTURELOAD**
- **D3DTRIANGLE**

Enumerated Types

This section contains information about the following enumerated types used with Direct3D Immediate Mode.

- **D3DANTIALIASMODE**
- **D3DBLEND**
- **D3DCMPFUNC**
- **D3DCULL**
- **D3DFILLMODE**
- **D3DFOGMODE**
- **D3DLIGHTTYPE**
- **D3DMATERIALCOLORSOURCE**
- **D3DPRIMITIVETYPE**
- **D3DRENDERSTATETYPE**
- **D3DSHADEMODE**

- **D3DSTATEBLOCKTYPE**
- **D3DSTENCILOP**
- **D3DTEXTUREADDRESS**
- **D3DTEXTUREMAGFILTER**
- **D3DTEXTUREMINFILTER**
- **D3DTEXTUREMIPFILTER**
- **D3DTEXTUREOP**
- **D3DTEXTURESTAGESTATETYPE**
- **D3DTEXTURETRANSFORMFLAGS**
- **D3DTRANSFORMSTATETYPE**
- **D3DVERTEXBLENDFLAGS**
- **D3DVERTEXTYPE**
- **D3DZBUFFERTYPE**

D3DANTIALIASMODE

The **D3DANTIALIASMODE** enumerated type defines the supported antialiasing mode for the **D3DRENDERSTATE_ANTIALIAS** value in the **D3DRENDERSTATETYPE** enumerated type. These values define the settings only for full-scene antialiasing (for more information, see Antialiasing).

```
typedef enum _D3DANTIALIASMODE {
    D3DANTIALIAS_NONE          = 0,
    D3DANTIALIAS_SORTDEPENDENT = 1,
    D3DANTIALIAS_SORTINDEPENDENT = 2,
    D3DANTIALIAS_FORCE_DWORD   = 0xffffffff,
} D3DANTIALIASMODE;
```

D3DANTIALIAS_NONE

No antialiasing is performed. This is the default setting.

D3DANTIALIAS_SORTDEPENDENT

Antialiasing is dependent on the sort order of the polygons (back to front or front to back). The application must draw polygons in the right order for antialiasing to occur.

D3DANTIALIAS_SORTINDEPENDENT

Antialiasing is not dependent on the sort order of the polygons.

D3DANTIALIAS_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

D3DBLEND

The **D3DBLEND** enumerated type defines the supported blend mode for the D3DRENDERSTATE_DESTBLEND values in the **D3DRENDERSTATETYPE** enumerated type. In the member descriptions that follow, the RGBA values of the source and destination are indicated by the subscripts *s* and *d*.

```
typedef enum _D3DBLEND {
    D3DBLEND_ZERO          = 1,
    D3DBLEND_ONE           = 2,
    D3DBLEND_SRCCOLOR      = 3,
    D3DBLEND_INVSRCOLOR    = 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;
```

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_INVSRCOLOR

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 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 **D3DRENDERSTATE_SRCBLEND** render state.

D3DBLEND_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in `d3dtypes.h`.

D3DCMPFUNC

The **D3DCMPFUNC** enumerated type defines the supported compare functions for the **D3DRENDERSTATE_ZFUNC**, **D3DRENDERSTATE_ALPHAFUNC**, and **D3DRENDERSTATE_STENCILFUNC** render states.

```
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;
```

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.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

D3DCULL

The **D3DCULL** enumerated type defines the supported culling modes used with the **D3DRENDERSTATE_CULLMODE** render state. These define how back faces are culled when rendering a geometry.

```
typedef enum _D3DCULL {
    D3DCULL_NONE = 1,
    D3DCULL_CW = 2,
    D3DCULL_CCW = 3,
    D3DCULL_FORCE_DWORD = 0x7fffffff,
} D3DCULL;
```

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.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DPRIMCAPS, D3DRENDERSTATETYPE

D3DFILLMODE

The **D3DFILLMODE** enumerated type contains constants describing the fill mode. These values are used by the D3DRENDERSTATE_FILLMODE render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DFILLMODE {
    D3DFILL_POINT    = 1,
    D3DFILL_WIREFRAME = 2,
    D3DFILL_SOLID    = 3,
    D3DFILL_FORCE_DWORD = 0x7fffffff,
} D3DFILLMODE;
```

D3DFILL_POINT

Fill points.

D3DFILL_WIREFRAME

Fill wireframes. This fill mode currently does not work for clipped primitives when you are using 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.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

D3DFOGMODE

The **D3DFOGMODE** enumerated type contains constants describing the fog mode. These values are used by the **D3DRENDERSTATE_FOGTABLEMODE** and **D3DRENDERSTATE_FOGVERTEXMODE** render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DFOGMODE {
    D3DFOG_NONE    = 0,
    D3DFOG_EXP     = 1,
    D3DFOG_EXP2    = 2,
    D3DFOG_LINEAR  = 3,
    D3DFOG_FORCE_DWORD = 0x7fffffff,
} D3DFOGMODE;
```

D3DFOG_NONE

No fog effect.

D3DFOG_EXP

The fog effect intensifies exponentially, according to the following formula:

$$f = \frac{1}{e^{d \times \text{density}}}$$

D3DFOG_EXP2

The fog effect intensifies exponentially with the square of the distance, according to the following formula:

$$f = \frac{1}{e^{(d \times \text{density})^2}}$$

D3DFOG_LINEAR

The fog effect intensifies linearly between the start and end points, according to the following formula:

$$f = \frac{\text{end} - d}{\text{end} - \text{start}}$$

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

In monochromatic (ramp) lighting mode, fog works properly only when the fog color is black. (If there is no lighting, any fog color works since, in this case, any fog color is effectively black.)

For more information, see Fog.

Note

Fog can be considered a measure of visibility—the lower the fog value produced by one of the fog equations, the less visible an object is.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

D3DLIGHTTYPE

The **D3DLIGHTTYPE** enumerated type defines the light type. This enumerated type is used with the **D3DLIGHT7** structure.

```
typedef enum _D3DLIGHTTYPE {
    D3DLIGHT_POINT      = 1,
    D3DLIGHT_SPOT       = 2,
    D3DLIGHT_DIRECTIONAL = 3,
    D3DLIGHT_FORCE_DWORD = 0x7fffffff,
} D3DLIGHTTYPE;
```

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 **D3DLIGHT7** 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.

Parallel-point lights are not supported in DirectX 7.0 and later. Therefore, this enumerated type does not contain the legacy D3DLIGHT_PARALLELPOINT member.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

D3DMATERIALCOLORSOURCE

The **D3DMATERIALCOLORSOURCE** enumerated type 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;
```

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

The values defined by this enumeration are used with the following render states:

- D3DRENDERSTATE_DIFFUSEMATERIALSOURCE
- D3DRENDERSTATE_SPECULARMATERIALSOURCE
- D3DRENDERSTATE_AMBIENTMATERIALSOURCE
- D3DRENDERSTATE_EMISSIVEMATERIALSOURCE

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3dtypes.h.

D3DPRIMITIVETYPE

The **D3DPRIMITIVETYPE** enumerated type lists the primitives supported by DrawPrimitive methods. This type was introduced in DirectX 5.0.

```
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;
```

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. Calls using this primitive type fail if the count is less than 3 or not evenly divisible by 3.

Backface culling is affected by the current winding-order render state.

D3DPT_TRIANGLESTRIP

Renders the vertices as a triangle strip. Calls using this primitive type fail if the count is less than 3. The backface-culling flag is automatically flipped on even-numbered triangles.

D3DPT_TRIANGLEFAN

Renders the vertices as a triangle fan. Calls using this primitive type fail if the count is less than 3.

D3DPT_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

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.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::DrawIndexedPrimitive,
IDirect3DDevice7::DrawIndexedPrimitiveStrided,
IDirect3DDevice7::DrawIndexedPrimitiveVB,
IDirect3DDevice7::DrawPrimitive, **IDirect3DDevice7::DrawPrimitiveStrided**,
IDirect3DDevice7::DrawPrimitiveVB, Primitive Types

D3DRENDERSTATETYPE

The **D3DRENDERSTATETYPE** enumerated type defines device render states that are set and retrieved through the **IDirect3DDevice7::SetRenderState** and **IDirect3DDevice7::GetRenderState** methods.

```
typedef enum _D3DRENDERSTATETYPE {
    D3DRENDERSTATE_ANTIALIAS        = 2,    //Antialiasing mode
    D3DRENDERSTATE_TEXTUREPERSPECTIVE = 4,    //Perspective
    correction
    D3DRENDERSTATE_ZENABLE          = 7,    //Enable z test
    D3DRENDERSTATE_FILLMODE         = 8,    //Fill mode
    D3DRENDERSTATE_SHADEMODE        = 9,    //Shade mode
    D3DRENDERSTATE_LINEPATTERN      = 10,   //Line pattern
}
```

D3DRENDERSTATE_ZWRITEENABLE = 14, //Enable z writes
 D3DRENDERSTATE_ALPHATESTENABLE = 15, //Enable alpha tests
 D3DRENDERSTATE_LASTPIXEL = 16, //Draw last pixel in a line
 D3DRENDERSTATE_SRCBLEND = 19, //Blend factor for source
 D3DRENDERSTATE_DESTBLEND = 20, //Blend factor for
 destination
 D3DRENDERSTATE_CULLMODE = 22, //Back-face culling mode
 D3DRENDERSTATE_ZFUNC = 23, //Z-comparison function
 D3DRENDERSTATE_ALPHAREF = 24, //Reference alpha value
 D3DRENDERSTATE_ALPHAFUNC = 25, //Alpha-comparison
 function
 D3DRENDERSTATE_DITHERENABLE = 26, //Enable dithering
 D3DRENDERSTATE_ALPHABLENDENABLE = 27, //Enable alpha
 blending
 D3DRENDERSTATE_FOGENABLE = 28, //Enable fog
 D3DRENDERSTATE_SPECULARENABLE = 29, //Enable specular
 highlights
 D3DRENDERSTATE_ZVISIBLE = 30, //Enable z-checking
 D3DRENDERSTATE_STIPPLEDALPHA = 33, //Enable stippled alpha
 D3DRENDERSTATE_FOGCOLOR = 34, //Fog color
 D3DRENDERSTATE_FOGTABLEMODE = 35, //Fog mode
 D3DRENDERSTATE_FOGTABLESTART = 36, //Fog table start (same
 as D3DRENDERSTATE_FOGSTART)
 D3DRENDERSTATE_FOGTABLEEND = 37, //Fog table end (same as
 D3DRENDERSTATE_FOGEND)
 D3DRENDERSTATE_FOGTABLEDENSITY = 38, //Fog table density
 (same as D3DRENDERSTATE_FOGDENSITY)
 D3DRENDERSTATE_FOGSTART = 36, //Fog start (for both vertex
 and pixel fog)
 D3DRENDERSTATE_FOGEND = 37, //Fog end (for both vertex
 and pixel fog)
 D3DRENDERSTATE_FOGDENSITY = 38, //Fog density (for both
 vertex and pixel fog)
 D3DRENDERSTATE_EDGEANTIALIAS = 40, //Antialias edges
 D3DRENDERSTATE_COLORKEYENABLE = 41, //Enable color-key
 transparency
 D3DRENDERSTATE_ZBIAS = 47, //Z-bias
 D3DRENDERSTATE_RANGEFOGENABLE = 48, //Enables range-
 based fog
 D3DRENDERSTATE_STENCILENABLE = 52, //Enable or disable
 stencil
 D3DRENDERSTATE_STENCILFAIL = 53, //Stencil operation
 D3DRENDERSTATE_STENCILZFAIL = 54, //Stencil operation
 D3DRENDERSTATE_STENCILPASS = 55, //Stencil operation
 D3DRENDERSTATE_STENCILFUNC = 56, //Stencil comparison
 function

```

    D3DRENDERSTATE_STENCILREF      = 57, //Reference value for
stencil test
    D3DRENDERSTATE_STENCILMASK    = 58, //Mask value used in
stencil test
    D3DRENDERSTATE_STENCILWRITEMASK = 59, //Stencil buffer write
mask
    D3DRENDERSTATE_TEXTUREFACTOR  = 60, //Texture factor
    D3DRENDERSTATE_WRAP0          = 128, //Wrap flags for 1st texture
coord set
    // Wrap render states 1 through 6 omitted here.
    D3DRENDERSTATE_WRAP7          = 135, //Wrap flags for last texture
coord set
    D3DRENDERSTATE_CLIPPING       = 136, //Enable or disable primitive
clipping
    D3DRENDERSTATE_LIGHTING       = 137, //Enable or disable lighting
    D3DRENDERSTATE_EXTENTS        = 138, //Enable or disable updating
screen extents
    D3DRENDERSTATE_AMBIENT        = 139, //Ambient color for scene
    D3DRENDERSTATE_FOGVERTEXMODE  = 140, //Fog mode for vertex
fog
    D3DRENDERSTATE_COLORVERTEX    = 141, //Enable or disable per-
vertex color
    D3DRENDERSTATE_LOCALVIEWER    = 142, //Enable or disable
perspective specular highlights
    D3DRENDERSTATE_NORMALIZENORMALS = 143, //Enable automatic
normalization of vertex normals
    D3DRENDERSTATE_COLORKEYBLENDENABLE = 144, //Enable or
disable alpha-blended color keying
    D3DRENDERSTATE_DIFFUSEMATERIALSOURCE = 145, //Location for
per-vertex diffuse color
    D3DRENDERSTATE_SPECULARMATERIALSOURCE = 146, //Location for
per-vertex specular color
    D3DRENDERSTATE_AMBIENTMATERIALSOURCE = 147, //Location for
per-vertex ambient color
    D3DRENDERSTATE_EMISSIVEMATERIALSOURCE = 148, //Location for
per-vertex emissive color
    D3DRENDERSTATE_VERTEXBLEND    = 151, //Multi-matrix vertex
blending mode
    D3DRENDERSTATE_CLIPPLANEENABLE = 152, //Enable one or
more user-defined clipping planes
    D3DRENDERSTATE_FORCE_DWORD    = 0x7fffffff,
} D3DRENDERSTATETYPE;
D3DRENDERSTATE_ANTI_ALIAS

```

One of the members of the **D3DANTI_ALIASMODE** enumerated type specifying the desired type of full-scene antialiasing. The default value is

D3DANTIALIAS_NONE. For more information, see Full-scene Antialiasing and Antialiasing States.

You can only enable full-scene antialiasing on devices that expose the D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT or D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT capabilities.

D3DRENDERSTATE_TEXTUREPERSPECTIVE

TRUE to enable perspective-correct texture mapping. (See perspective correction.) The default value is TRUE. For interfaces earlier than **IDirect3DDevice3**, the default is FALSE. For more information, see Texture Perspective State.

D3DRENDERSTATE_ZENABLE

Depth-buffering state as one of the members 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 buffer is attached to the render-target surface, and D3DZB_FALSE otherwise.

D3DRENDERSTATE_FILLMODE

One or more members of the **D3DFILLMODE** enumerated type. The default value is D3DFILL_SOLID.

D3DRENDERSTATE_SHADEMODE

One or more members of the **D3DSHADEMODE** enumerated type. The default value is D3DSHADE_GOURAUD.

D3DRENDERSTATE_LINEPATTERN

The **D3DLINEPATTERN** structure. The default values are 0 for **wRepeatPattern**, and 0 for **wLinePattern**.

D3DRENDERSTATE_ZWRITEENABLE

TRUE to enable writes 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 D3DRENDERSTATE_ZFUNC (assuming that depth buffering is taking place), but depth values are not written to the buffer.

D3DRENDERSTATE_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 D3DRENDERSTATE_ALPHAFUNC render state. When this mode is enabled, alpha blending occurs only if the test succeeds.

D3DRENDERSTATE_LASTPIXEL

FALSE to enable drawing the last pixel in a line or triangle. The default value is TRUE.

D3DRENDERSTATE_SRCBLEND

One of the members of the **D3DBLEND** enumerated type. The default value is **D3DBLEND_ONE**.

D3DRENDERSTATE_DESTBLEND

One of the members of the **D3DBLEND** enumerated type. The default value is **D3DBLEND_ZERO**.

D3DRENDERSTATE_CULLMODE

Specifies how back-facing triangles are to be culled, if at all. This can be set to one of the members of the **D3DCULL** enumerated type. The default value is **D3DCULL_CCW**.

D3DRENDERSTATE_ZFUNC

One of the members 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, since there is no need to filter and modulate the texture if the pixel is not going to be rendered.

D3DRENDERSTATE_ALPHAREF

Value specifying 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.

D3DRENDERSTATE_ALPHAFUNC

One of the members 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.

D3DRENDERSTATE_DITHERENABLE

TRUE to enable dithering. The default value is **FALSE**.

D3DRENDERSTATE_ALPHABLENDENABLE

TRUE to enable alpha-blended transparency. The default value is **FALSE**. This member supersedes the **D3DRENDERSTATE_BLENDENABLE** render state; for more information, see remarks.

Prior to DirectX 5.0, the software rasterizers used this render state to toggle both color keying and alpha blending. Currently, use the **D3DRENDERSTATE_COLORKEYENABLE** render state to toggle color keying. (Hardware rasterizers have always used the **D3DRENDERSTATE_BLENDENABLE** render state only for toggling alpha blending.)

The type of alpha blending is determined by the **D3DRENDERSTATE_SRCBLEND** and **D3DRENDERSTATE_DESTBLEND** render states. **D3DRENDERSTATE_ALPHABLENDENABLE**, with **D3DRENDERSTATE_COLORKEYENABLE**, allows fine blending control.

D3DRENDERSTATE_ALPHABLENDENABLE does not affect the texture-blending modes specified by the **D3DTEXTUREBLEND** enumerated type. Texture blending is logically done well before the D3DRENDERSTATE_ALPHABLENDENABLE part of the pixel pipeline. The only interaction between the two is that the alpha portions remaining in the polygon after the **D3DTEXTUREBLEND** phase can be used in the D3DRENDERSTATE_ALPHABLENDENABLE phase to govern interaction with the content in the frame buffer.

Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **D3DDEVICEDESC7** structure to find out whether this render state is supported.

D3DRENDERSTATE_FOGENABLE

TRUE to enable fog blending. The default value is FALSE. For more information, see Fog Blending and Fog.

D3DRENDERSTATE_SPECULARENABLE

TRUE to enable specular highlights. The default value is FALSE. For interfaces earlier than **IDirect3DDevice3**, the default value is TRUE.

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.

D3DRENDERSTATE_ZVISIBLE

Not supported.

D3DRENDERSTATE_STIPPLEDALPHA

TRUE to enable stippled alpha. The default value is FALSE.

Current software rasterizers ignore this render state. Use the D3DPSHADECAPS_ALPHAFLATSTIPPLED flag in the **D3DPRIMCAPS** structure to discover whether the current hardware supports this render state.

D3DRENDERSTATE_FOGCOLOR

Value whose type is **D3DCOLOR**. The default value is 0. For more information, see Fog Color.

D3DRENDERSTATE_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, see Pixel Fog.

D3DRENDERSTATE_FOGTABLESTART

Maps to D3DRENDERSTATE_FOGSTART.

D3DRENDERSTATE_FOGTABLEEND

Maps to D3DRENDERSTATE_FOGEND.

D3DRENDERSTATE_FOGTABLEDENSITY

This render state maps to the D3DRENDERSTATE_FOGDENSITY render state.

D3DRENDERSTATE_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 **IDirect3DDevice7::SetRenderState** and **IDirect3DDevice7::SetRenderState** accept DWORD values, your application will must cast a variable that contains the value, as shown in the following code example.

```
lpd3dDevice->SetRenderState( D3DRENDERSTATE_FOGSTART, *((LPDWORD)
(&fFogStart)) );
```

D3DRENDERSTATE_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 **IDirect3DDevice7::SetRenderState** and **IDirect3DDevice7::SetRenderState** accept DWORD values, your application will must cast a variable that contains the value, as shown in the following code example.

```
lpd3dDevice->SetRenderState( D3DRENDERSTATE_FOGEND, *((LPDWORD)
(&fFogEnd)) );
```

D3DRENDERSTATE_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 **IDirect3DDevice7::SetRenderState** and **IDirect3DDevice7::SetRenderState** accept DWORD values, your application will must cast a variable that contains the value, as shown in the following code example.

```
lpd3dDevice->SetRenderState( D3DRENDERSTATE_FOGDENSITY, *((LPDWORD)
(&fFogDensity)) );
```

D3DRENDERSTATE_EDGEANTIALIAS

TRUE to antialias lines forming the convex outline of objects. The default value is FALSE. For more information, see Edge Antialiasing and Antialiasing States. 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 only enable edge antialiasing on devices that expose the `D3DPRASERCAPS_ANTIALIASEDGES` capability.

`D3DRENDERSTATE_COLORKEYENABLE`

TRUE to enable color-keyed transparency. The default value is FALSE. Use this render state with `D3DRENDERSTATE_ALPHABLENDENABLE` to implement fine blending control.

Applications should check the `D3DDEVCAPS_DRAWPRIMTLVERTEX` flag in the **`D3DDEVICEDESC7`** structure to find out whether this render state is supported.

When color-keyed transparency is enabled, only texture surfaces that were created with the `DDSD_CKSRCLT` flag are affected. Surfaces that were created without the `DDSD_CKSRCLT` flag exhibit color-keyed transparency effects.

`D3DRENDERSTATE_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*.

`D3DRENDERSTATE_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 foggiest of peripheral objects change as the viewer's eye moves—in this case, the depth changes, and the range remains constant.

This render state works only with **`D3DVERTEX`** vertices. When you specify **`D3DLVERTEX`** or **`D3DTLVERTEX`** vertices, the F (fog) component of the RGBF fog value should already be corrected for range.

Since 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*.

`D3DRENDERSTATE_STENCILENABLE`

TRUE to enable stencil, or FALSE to disable stencil. The default value is FALSE. For more information, see *Stencil Buffers*.

`D3DRENDERSTATE_STENCILFAIL`

Stencil operation to perform if the stencil test fails. This can be one of the members of the **`D3DSTENCILOP`** enumerated type. The default value is `D3DSTENCILOP_KEEP`. For more information, see *Stencil Buffers*.

`D3DRENDERSTATE_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. For more information, see Stencil Buffers.

D3DRENDERSTATE_STENCILPASS

Stencil operation to perform if both the stencil and the depth (z) tests pass. This can be one of the members of the **D3DSTENCILOP** enumerated type. The default value is D3DSTENCILOP_KEEP. For more information, see Stencil Buffers.

D3DRENDERSTATE_STENCILFUNC

Comparison function for the stencil test. This can be one of the members 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 only applies to the bits in the reference value and stencil buffer entry that are set in the stencil mask (set by the D3DRENDERSTATE_STENCILMASK render state). If TRUE, the stencil test passes.

D3DRENDERSTATE_STENCILREF

Integer reference value for the stencil test. The default value is 0.

D3DRENDERSTATE_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.

D3DRENDERSTATE_STENCILWRITEMASK

Write mask applied to values written into the stencil buffer. The default mask is 0xFFFFFFFF.

D3DRENDERSTATE_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.

D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7

Texture-wrapping behavior for multiple sets of texture coordinates. Valid values for these render states can be any combination of the D3DWRAPCOORD_0, D3DWRAPCOORD_1, D3DWRAPCOORD_2, and D3DWRAPCOORD_3 flags. These cause the system to wrap in the direction of the first, second, third, and fourth dimensions (sometimes referred to as 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). For more information, see Texture Wrapping.

D3DRENDERSTATE_CLIPPING

TRUE to enable primitive clipping by Direct3D, or FALSE to disable it. The default value is TRUE.

D3DRENDERSTATE_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.

D3DRENDERSTATE_EXTENTS

TRUE to cause the system to update the screen extents for each rendering call, or FALSE to disable screen extent updates. The default value is FALSE.

D3DRENDERSTATE_AMBIENT

Ambient light color. This value is of type **D3DCOLOR**. The default value is 0. This render state is analogous to the legacy D3DLIGHTSTATE_AMBIENT lighting state.

D3DRENDERSTATE_FOGVERTEXMODE

Fog formula to be used for vertex fog. Set to one of the members of the **D3DFOGMODE** enumerated type. The default value is D3DFOG_NONE. For more information, see Vertex Fog. This render state is analogous to the legacy D3DLIGHTSTATE_FOGVERTEXMODE lighting state.

D3DRENDERSTATE_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. This render state is analogous to the legacy D3DLIGHTSTATE_COLORVERTEX lighting state.

For more information, see the following render states:

- D3DRENDERSTATE_DIFFUSEMATERIALSOURCE
- D3DRENDERSTATE_SPECULARMATERIALSOURCE
- D3DRENDERSTATE_AMBIENTMATERIALSOURCE
- D3DRENDERSTATE_EMISSIVEMATERIALSOURCE

D3DRENDERSTATE_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.

D3DRENDERSTATE_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.

D3DRENDERSTATE_COLORKEYBLENDENABLE

TRUE to enable alpha-blended colorkeying, or FALSE to disable it. Alpha-blended colorkey mode sets the alpha and color of matching texture samples to 0. The application must set the alpha-test and alpha-blend modes to achieve the desired discard and/or alpha-blended result.

D3DRENDERSTATE_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 D3DRENDERSTATE_COLORVERTEX render state is set to TRUE.

D3DRENDERSTATE_SPECULARMATERIALSOURCE

Specular color source for lighting calculations. Valid values are members of the **D3DMATERIALCOLORSOURCE** enumerated type. The default value is **D3DMCS_COLOR2**.

D3DRENDERSTATE_AMBIENTMATERIALSOURCE

Ambient color source for lighting calculations. Valid values are members of the **D3DMATERIALCOLORSOURCE** enumerated type. The default value is **D3DMCS_COLOR2**.

D3DRENDERSTATE_EMISSIVEMATERIALSOURCE

Emissive color source for lighting calculations. Valid values are members of the **D3DMATERIALCOLORSOURCE** enumerated type. The default value is **D3DMCS_MATERIAL**.

D3DRENDERSTATE_VERTEXBLEND

Number of matrices to be used to perform geometry blending, if any. Valid values are members of the **D3DVERTEXBLEND_FLAGS** enumerated type. The default value is **D3DVBLEND_DISABLE**. For more information see Geometry Blending.

D3DRENDERSTATE_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 declared in the **D3dtypes.h** header file to provide a convenient way to enable clipping planes.

D3DRENDERSTATE_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

The **D3DRENDERSTATE_BLENDENABLE** member was superseded by the **D3DRENDERSTATE_ALPHABLENDENABLE** member. Its name was changed to make its meaning more explicit. To maintain compatibility with previous applications, the **D3DRENDERSTATE_BLENDENABLE** constant is declared as equivalent to **D3DRENDERSTATE_ALPHABLENDENABLE**:

```
#define D3DRENDERSTATE_BLENDENABLE D3DRENDERSTATE_ALPHABLENDENABLE
```

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 **D3DRENDERSTATE_WRAP n** state values). Add the **D3DRENDERSTATE_WRAPBIAS** value to the zero-based index of a texture coordinate set to calculate the **D3DRENDERSTATE_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 = lpD3DDevice->SetRenderState(
    dwIndex + D3DRENDERSTATE_WRAPBIAS,
    D3DWRAP_0 | D3DWRAP1);

// If dwIndex is 3, the value that results from
// the addition equals D3DRENDERSTATE_WRAP3 (131).
```

The declaration of **D3DRENDERSTATETYPE** in D3dtypes.h contains legacy render states not supported by DirectX 7.0 interfaces. For more information, see Obsolete Render States.

```
D3DRENDERSTATE_TEXTUREHANDLE    = 1, //Texture handle
D3DRENDERSTATE_TEXTUREADDRESS  = 3, //Texture address
D3DRENDERSTATE_WRAPU           = 5, //Wrap in u direction
D3DRENDERSTATE_WRAPV           = 6, //Wrap in v direction
D3DRENDERSTATE_MONOENABLE       = 11, //Enable mono rendering
D3DRENDERSTATE_ROP2             = 12, //Raster operation
D3DRENDERSTATE_PLANEMASK       = 13, //Physical plane mask
D3DRENDERSTATE_TEXTUREMAG       = 17, //Superseded
D3DRENDERSTATE_TEXTUREMIN       = 18, //Superseded
D3DRENDERSTATE_TEXTUREMAPBLEND = 21, //Blend mode for map
D3DRENDERSTATE_SUBPIXEL        = 31, //Enable subpixel correction
D3DRENDERSTATE_SUBPIXELX       = 32, //Enable x subpixel correction
D3DRENDERSTATE_STIPPLEENABLE    = 39, //Enable stippling
D3DRENDERSTATE_BORDERCOLOR     = 43, //Border color
D3DRENDERSTATE_TEXTUREADDRESSU = 44, //U texture address mode
D3DRENDERSTATE_TEXTUREADDRESSV = 45, //V texture address mode
D3DRENDERSTATE_MIPMAPLODBIAS    = 46, //Mipmap LOD bias
D3DRENDERSTATE_ANISOTROPY       = 49, //Maximum anisotropy
D3DRENDERSTATE_FLUSHBATCH       = 50, //Explicit flush for DP batching (DX5 Only)
D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT=51, //Enable sort-independent
transparency
D3DRENDERSTATE_STIPPLEPATTERN00 = 64, //First line of stipple pattern
// Stipple patterns 01 through 30 omitted here.
D3DRENDERSTATE_STIPPLEPATTERN31 = 95, //Last line of stipple pattern
```

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::SetRenderState, **IDirect3DDevice7::GetRenderState**, Render States

D3DSHADEMODE

The **D3DSHADEMODE** enumerated type describes the supported shading mode for the **D3DRENDERSTATE_SHADEMODE** render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DSHADEMODE {
    D3DSHADE_FLAT      = 1,
    D3DSHADE_GOURAUD   = 2,
    D3DSHADE_PHONG     = 3,
    D3DSHADE_FORCE_DWORD = 0x7fffffff,
} D3DSHADEMODE;
```

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.

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 currently supported.

D3DSHADE_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DRENDERSTATETYPE

D3DSTATEBLOCKTYPE

The **D3DSTATEBLOCK** enumerated type defines logical groups of device states, for use with the **IDirect3DDevice7::CreateStateBlock** method.

```
typedef enum _D3DSTATEBLOCKTYPE{
    D3DSBT_ALL          = 1,
    D3DSBT_PIXELSTATE   = 2,
    D3DSBT_VERTEXSTATE  = 3,
    D3DSBT_FORCE_DWORD  = 0xffffffff
} D3DSTATEBLOCKTYPE;
```

D3DSBT_ALL

Capture all device states.

D3DSBT_PIXELSTATE

Capture only pixel-related device states.

D3DSBT_VERTEXSTATE

Capture only vertex-related device states.

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. For information about the states defined by each group, see [Creating Predefined State Blocks](#).

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in `d3dtypes.h`.

See Also

IDirect3DDevice7::CreateStateBlock

D3DSTENCILOP

The **D3DSTENCILOP** enumerated type describes the stencil operations for the **D3DRENDERSTATE_STENCILFAIL**, **D3DRENDERSTATE_STENCILZFAIL**, and **D3DRENDERSTATE_STENCILPASS** render states.

```
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;
```

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 0.

D3DSTENCILOP_INVERT

Invert the bits in the stencil-buffer entry.

D3DSTENCILOP_INCR

Increment the stencil-buffer entry, wrapping to 0 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 0.

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.
Version: Requires DirectX 6.0 or later.
Header: Declared in d3dtypes.h.

See Also

D3DRENDERSTATETYPE, Stencil Buffers

D3DTEXTUREADDRESS

The **D3DTEXTUREADDRESS** enumerated type describes the supported texture-addressing modes when setting them with the **D3DTSS_ADDRESS**, **D3DTSS_ADDRESSU**, and **D3DTSS_ADDRESSV** texture-stage states.

```
typedef enum _D3DTEXTUREADDRESS {
    D3DTEXTUREADDRESS_WRAP      = 1,
    D3DTEXTUREADDRESS_MIRROR    = 2,
    D3DTEXTUREADDRESS_CLAMP     = 3,
    D3DTEXTUREADDRESS_BORDER    = 4,
    D3DTEXTUREADDRESS_FORCE_DWORD = 0xffffffff,
} D3DTEXTUREADDRESS;
```

D3DTEXTUREADDRESS_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.

D3DTEXTUREADDRESS_MIRROR

Similar to **D3DTEXTUREADDRESS_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.

D3DTEXTUREADDRESS_CLAMP

Texture coordinates outside the range [0.0, 1.0] are set to the texture color at 0.0 or 1.0, respectively.

D3DTEXTUREADDRESS_BORDER

Texture coordinates outside the range [0.0, 1.0] are set to the border color, which is a new render state corresponding to **D3DRENDERSTATE_BORDERCOLOR** in the **D3DRENDERSTATETYPE** enumerated type.

D3DTEXTUREADDRESS_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

For information about using the **D3DRENDERSTATE_WRAPU** and **D3DRENDERSTATE_WRAPV** render states, see Textures.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DRENDERSTATETYPE

D3DTEXTUREMAGFILTER

The **D3DTEXTUREMAGFILTER** enumerated type defines texture-magnification filtering modes for a texture stage.

```
typedef enum _D3DTEXTUREMAGFILTER {
    D3DTFG_POINT      = 1,
    D3DTFG_LINEAR     = 2,
    D3DTFG_FLATCUBIC  = 3,
    D3DTFG_GAUSSIANCUBIC = 4,
    D3DTFG_ANISOTROPIC = 5,
    D3DTFG_FORCE_DWORD = 0x7fffffff,
} D3DTEXTUREMAGFILTER;
```

D3DTFG_POINT

Point filtering. The texel with coordinates nearest to the desired pixel value is used.

D3DTFG_LINEAR

Bilinear interpolation filtering. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

D3DTFG_FLATCUBIC

Not currently supported; do not use.

D3DTFG_GAUSSIANCUBIC

Not currently supported; do not use.

D3DTFG_ANISOTROPIC

Anisotropic texture filtering. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

D3DTFG_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Set a texture stage's magnification filter by calling the **IDirect3DDevice7::SetTextureStageState** method with the D3DTSS_MAGFILTER value as the second parameter and one of the members of this enumeration as the third parameter.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DTEXTUREMINFILTER, D3DTEXTUREMIPFILTER, Texture Filtering

D3DTEXTUREMINFILTER

The **D3DTEXTUREMINFILTER** enumerated type defines texture-minification filtering modes for a texture stage.

```
typedef enum _D3DTEXTUREMINFILTER {
    D3DTFN_POINT      = 1,
    D3DTFN_LINEAR     = 2,
    D3DTFN_ANISOTROPIC = 3,
    D3DTFN_FORCE_DWORD = 0x7fffffff,
} D3DTEXTUREMINFILTER;
```

D3DTFN_POINT

Point filtering. The texel with coordinates nearest to the desired pixel value is used.

D3DTFN_LINEAR

Bilinear interpolation filtering. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

D3DTFN_ANISOTROPIC

Anisotropic texture filtering. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

D3DTFN_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Set a texture stage's magnification filter by calling the **IDirect3DDevice7::SetTextureStageState** method with the D3DTSS_MINFILTER value as the second parameter and one of the members of this enumeration as the third parameter.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DTEXTUREMAGFILTER, D3DTEXTUREMIPFILTER, Texture Filtering

D3DTEXTUREMIPFILTER

The **D3DTEXTUREMIPFILTER** enumerated type defines texture mipmap filtering modes for a texture stage.

```
typedef enum _D3DTEXTUREMIPFILTER {
    D3DTFP_NONE      = 1,
    D3DTFP_POINT     = 2,
    D3DTFP_LINEAR    = 3,
    D3DTFP_FORCE_DWORD = 0x7fffffff,
} D3DTEXTUREMIPFILTER;
```

D3DTFP_NONE

Mipmapping disabled. The rasterizer should use the magnification filter instead.

D3DTFP_POINT

Nearest-point mipmap filtering. The rasterizer uses the color from the texel of the nearest mipmap texture.

D3DTFP_LINEAR

Trilinear mipmap interpolation. The rasterizer linearly interpolates pixel color, using the texels of the two nearest mipmap textures.

D3DTFP_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

Set a texture stage's magnification filter by calling the **IDirect3DDevice7::SetTextureStageState** method with the D3DTSS_MIPFILTER value as the second parameter and one of the members of this enumeration as the third parameter.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Windows CE: Unsupported.
Version: Requires DirectX 6.0 or later.
Header: Declared in d3dtypes.h.

See Also

D3DTEXTUREMAGFILTER, D3DTEXTUREMIPFILTER, Texture Filtering

D3DTEXTUREOP

The **D3DTEXTUREOP** enumerated type defines per-stage texture-blending operations. The members of this type are used when setting color or alpha operations by using the D3DTSS_COLOROP or D3DTSS_ALPHAOP values with the **IDirect3DDevice7::SetTextureStageState** method.

```
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_FORCE_DWORD = 0x7fffffff,
} D3DTEXTUREOP;

```

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.

$$S_{\text{RGBA}} = \text{Arg1}$$

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.

$$S_{\text{RGBA}} = \text{Arg2}$$

Modulation members

D3DTOP_MODULATE

Multiply the components of the arguments together.

$$S_{\text{RGBA}} = \text{Arg1} \times \text{Arg2}$$

D3DTOP_MODULATE2X

Multiply the components of the arguments, and shift the products to the left 1 bit (effectively multiplying them by 2) for brightening.

$$S_{\text{RGBA}} = (\text{Arg1} \times \text{Arg2}) \ll 1$$

D3DTOP_MODULATE4X

Multiply the components of the arguments, and shift the products to the left 2 bits (effectively multiplying them by 4) for brightening.

$$S_{\text{RGBA}} = (\text{Arg1} \times \text{Arg2}) \ll 2$$

Addition and subtraction members**D3DTOP_ADD**

Add the components of the arguments.

$$S_{\text{RGBA}} = \text{Arg1} + \text{Arg2}$$

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.

$$S_{\text{RGBA}} = \text{Arg1} + \text{Arg2} - 0.5$$

D3DTOP_ADDSIGNED2X

Add the components of the arguments with a –0.5 bias, and shift the products to the left 1 bit.

$$S_{\text{RGBA}} = (\text{Arg1} + \text{Arg2} - 0.5) \ll 1$$

D3DTOP_SUBTRACT

Subtract the components of the second argument from those of the first argument.

$$S_{\text{RGBA}} = \text{Arg1} - \text{Arg2}$$

D3DTOP_ADDSMOOTH

Add the first and second arguments, then subtract their product from the sum.

$$\begin{aligned} S_{\text{RGBA}} &= \text{Arg1} + \text{Arg2} - \text{Arg1} \times \text{Arg2} \\ &= \text{Arg1} + \text{Arg2} (1 - \text{Arg1}) \end{aligned}$$

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 D3DRENDERSTATE_TEXTUREFACTOR render state, or the alpha taken from the previous texture stage (D3DTOP_BLENDCURRENTALPHA).

$$S_{\text{RGBA}} = \text{Arg1} \times (\text{Alpha}) + \text{Arg2} \times (1 - \text{Alpha})$$

D3DTOP_BLENDTEXTUREALPHAPM

Linearly blend a texture stage that uses a premultiplied alpha.

$$S_{\text{RGBA}} = \text{Arg1} + \text{Arg2} \times (1 - \text{Alpha})$$

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).

$$S_{\text{RGBA}} = \text{Arg1}_{\text{RGB}} + \text{Arg1}_{\text{A}} \times \text{Arg2}_{\text{RGB}}$$

D3DTOP_MODULATECOLOR_ADDALPHA

Modulate the arguments; then add the alpha of the first argument. This operation is supported only for color operations (D3DTSS_COLOROP).

$$S_{\text{RGBA}} = \text{Arg1}_{\text{RGB}} \times \text{Arg2}_{\text{RGB}} + \text{Arg1}_{\text{A}}$$

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).

$$S_{\text{RGBA}} = (1 - \text{Arg1}_{\text{A}}) \times \text{Arg2}_{\text{RGB}} + \text{Arg1}_{\text{RGB}}$$

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).

$$S_{\text{RGBA}} = (1 - \text{Arg1}_{\text{RGB}}) \times \text{Arg2}_{\text{RGB}} + \text{Arg1}_{\text{A}}$$

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.

$$S_{\text{RGBA}} = (\text{Arg1}_{\text{R}} \times \text{Arg2}_{\text{R}} + \text{Arg1}_{\text{G}} \times \text{Arg2}_{\text{G}} + \text{Arg1}_{\text{B}} \times \text{Arg2}_{\text{B}})$$

Miscellaneous member**D3DTOP_FORCE_DWORD**

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

In the 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 $Arg1_A$.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::GetTextureStageState,

IDirect3DDevice7::SetTextureStageState, D3DTEXTURESTAGESTATETYPE

D3DTEXTURESTAGESTATETYPE

The **D3DTEXTURESTAGESTATETYPE** enumerated type defines texture stage states. Members of this enumerated type are used with the

IDirect3DDevice7::GetTextureStageState and

IDirect3DDevice7::SetTextureStageState methods to retrieve and set texture state values.

```
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_ADDRESS          = 12,
    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_FORCE_DWORD  = 0x7fffffff,
} D3DTEXTURESTAGESTATETYPE;

```

D3DTSS_COLOROP

The texture-stage state is a texture color blending operation identified by one of the members 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.

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.

D3DTSS_ALPHAOP

The texture-stage state is a texture alpha blending operation identified by one of the members 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.

For DirectX 6.0, the default behavior for this state is not described as a texture operation. Rather, the alpha blending behavior matches that of the D3DRENDERSTATE_TEXTUREMAPBLEND render state with a value of D3DTBLEND_MODULATE.

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.

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.

D3DTSS_BUMPENVMAT00

The texture-stage state is a **D3DVALUE** for the [0][0] coefficient in a bump-mapping matrix. The default value is 0.

D3DTSS_BUMPENVMAT01

The texture-stage state is a **D3DVALUE** for the [0][1] coefficient in a bump-mapping matrix. The default value is 0.

D3DTSS_BUMPENVMAT10

The texture-stage state is a **D3DVALUE** for the [1][0] coefficient in a bump-mapping matrix. The default value is 0.

D3DTSS_BUMPENVMAT11

The texture-stage state is a **D3DVALUE** for the [1][1] coefficient in a bump-mapping matrix. The default value is 0.

D3DTSS_TEXCOORDINDEX

Index of the texture coordinate set to use with this texture stage. The default index is 0. 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 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_ADDRESS

Member of the **D3DTEXTUREADDRESS** enumerated type. Selects the texture-addressing method for both the u and v coordinates. The default is D3DTEXTUREADDRESS_WRAP.

D3DTSS_ADDRESSU

Member of the **D3DTEXTUREADDRESS** enumerated type. Selects the texture-addressing method for the u coordinate. The default is D3DTEXTUREADDRESS_WRAP.

D3DTSS_ADDRESSV

Member of the **D3DTEXTUREADDRESS** enumerated type. Selects the texture-addressing method for the v coordinate. The default value is D3DTEXTUREADDRESS_WRAP.

D3DTSS_BORDERCOLOR

D3DCOLOR value that describes the color to be used for rasterizing texture coordinates outside the [0.0,1.0] range. The default color is 0x00000000.

D3DTSS_MAGFILTER

Member of the **D3DTEXTUREMAGFILTER** enumerated type that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFG_POINT.

D3DTSS_MINFILTER

Member of the **D3DTEXTUREMINFILTER** enumerated type that indicates the texture minification filter to be used when rendering the texture onto primitives. The default value is D3DTFN_POINT.

D3DTSS_MIPFILTER

Member of the **D3DTEXTUREMIPFILTER** enumerated type that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFP_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. Values for this state are floating-point values. Because the **IDirect3DDevice7::SetTextureStageState** and **IDirect3DDevice7::GetTextureStageState** accept DWORD values, your application will must cast a variable that contains the value, as shown in the following code example.

```
lpd3dDevice->SetTextureStageState( D3DTSS_MIPMAPLODBIAS, *((LPDWORD)
(&fBias)) );
```

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 does not want to 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

D3DVALUE scale for bump-map luminance. The default value is 0.

D3DTSS_BUMPENVLOFFSET

D3DVALUE offset for bump-map luminance. The default value is 0.

D3DTSS_TEXTURETRANSFORMFLAGS

Member of the **D3DTEXTURETRANSFORMFLAGS** enumeration that controls the transformation of texture coordinates for this texture stage. The default value is D3DTTFF_DISABLE.

D3DTSS_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dtypes.h.

D3DTEXTURETRANSFORMFLAGS

The **D3DTEXTURETRANSFORMFLAGS** enumerated type defines values used with the D3DTSS_TEXTURETRANSFORMFLAGS texture-stage state.

```
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;
```

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 will be 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.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 7.0.

Header: Declared in d3dtypes.h.

See Also

D3DTSS_TEXTURETRANSFORMFLAGS, Texture Coordinate Transformations

D3DTRANSFORMSTATETYPE

The **D3DTRANSFORMSTATETYPE** enumerated type describes the transformation state values used with the **IDirect3DDevice7::GetTransform** and **IDirect3DDevice7::SetTransform** methods.

```
typedef enum _D3DTRANSFORMSTATETYPE {
    D3DTRANSFORMSTATE_WORLD      = 1,
    D3DTRANSFORMSTATE_VIEW       = 2,
    D3DTRANSFORMSTATE_PROJECTION = 3,
    D3DTRANSFORMSTATE_WORLD1     = 4,
    D3DTRANSFORMSTATE_WORLD2     = 5,
    D3DTRANSFORMSTATE_WORLD3     = 6,
    D3DTRANSFORMSTATE_TEXTURE0   = 16,
    D3DTRANSFORMSTATE_TEXTURE1   = 17,
    D3DTRANSFORMSTATE_TEXTURE2   = 18,
    D3DTRANSFORMSTATE_TEXTURE3   = 19,
    D3DTRANSFORMSTATE_TEXTURE4   = 20,
    D3DTRANSFORMSTATE_TEXTURE5   = 21,
    D3DTRANSFORMSTATE_TEXTURE6   = 22,
    D3DTRANSFORMSTATE_TEXTURE7   = 23,
    D3DTRANSFORMSTATE_FORCE_DWORD = 0x7fffffff,
} D3DTRANSFORMSTATETYPE;
```

D3DTRANSFORMSTATE_WORLD

Identifies the transformation matrix being set as the world transformation matrix. The default value is NULL (the identity matrix).

D3DTRANSFORMSTATE_WORLD1, D3DTRANSFORMSTATE_WORLD2,
D3DTRANSFORMSTATE_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.

D3DTRANSFORMSTATE_VIEW

Identifies the transformation matrix being set as the view transformation matrix. The default value is NULL (the identity matrix).

D3DTRANSFORMSTATE_PROJECTION

Identifies the transformation matrix being set as the projection transformation matrix. The default value is NULL (the identity matrix).

D3DTRANSFORMSTATE_TEXTURE0 through
D3DTRANSFORMSTATE_TEXTURE7

Identifies the transformation matrix being set for the specified texture stage.

D3DTRANSFORMSTATE_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::GetTransform, IDirect3DDevice7::SetTransform

D3DVERTEXBLEND_FLAGS

The **D3DVERTEXBLEND_FLAGS** enumerated type defines flags used to control the number or matrices that the system applies when performing multimatrix vertex blending. Members of this type are used with the D3DRENDERSTATE_VERTEXBLEND render state.

```
typedef enum _D3DVERTEXBLEND_FLAGS {
    D3DVBLEND_DISABLE = 0,
    D3DVBLEND_1WEIGHT = 1,
    D3DVBLEND_2WEIGHTS = 2,
    D3DVBLEND_3WEIGHTS = 3,
} D3DVERTEXBLEND_FLAGS;
```

Members

D3DVBLEND_DISABLE

Disable vertex blending; only apply the world matrix specified by the D3DTRANSFORMSTATE_WORLD transformation state.

D3DVBLEND_1WEIGHT

Enable vertex blending between the two matrices set by the D3DTRANSFORMSTATE_WORLD and D3DTRANSFORMSTATE_WORLD1 transformation states.

D3DVBLEND_2WEIGHTS

Enable vertex blending between the three matrices set by the D3DTRANSFORMSTATE_WORLD, D3DTRANSFORMSTATE_WORLD1, and D3DTRANSFORMSTATE_WORLD2 transformation states.

D3DVBLEND_3WEIGHTS

Enable vertex blending between the four matrices set by the D3DTRANSFORMSTATE_WORLD, D3DTRANSFORMSTATE_WORLD1, D3DTRANSFORMSTATE_WORLD2, and D3DTRANSFORMSTATE_WORLD3 transformation states.

Remarks

Geometry blending (multimatrix vertex blending) requires that your application use a vertex format that has blending (beta) weights for each vertex.

See Also

D3DRENDERSTATETYPE, D3DTRANSFORMSTATETYPE, IDirect3DDevice7::SetTransform, Geometry Blending

D3DVERTEXTYPE

The **D3DVERTEXTYPE** enumerated type lists the vertex types that are supported by the legacy **IDirect3DDevice3**, **IDirect3DDevice2**, and **IDirect3DDevice** interfaces. If your application uses interfaces later than **IDirect3DDevice3**, such as **IDirect3DDevice7**, the **D3DVERTEXTYPE** enumerated type is superseded by flexible vertex format flags.

```
typedef enum _D3DVERTEXTYPE {
    D3DVT_VERTEX      = 1,
    D3DVT_LVERTEX     = 2,
    D3DVT_TLVERTEX    = 3,
    D3DVT_FORCE_DWORD = 0x7fffffff,
} D3DVERTEXTYPE;
```

D3DVT_VERTEX

All the vertices in the array are of the **D3DVERTEX** type. This setting causes transformation, lighting, and clipping to be applied to the primitive as it is rendered.

D3DVT_LVERTEX

All the vertices in the array are of the **D3DLVERTEX** type. When used with this option, the primitive has transformations applied during rendering.

D3DVT_TLVERTEX

All the vertices in the array are of the **D3DTLVERTEX** type. Rasterization is applied only to this data.

D3DVT_FORCE_DWORD

Forces this enumeration to compile to 32 bits in size. This value is not used.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

Vertex Formats

D3DZBUFFERTYPE

The **D3DZBUFFERTYPE** enumerated type describes depth-buffer formats for use with the D3DRENDERSTATE_ZENABLE render state.

```
typedef enum _D3DZBUFFERTYPE {
    D3DZB_FALSE      = 0,
    D3DZB_TRUE       = 1,
    D3DZB_USEW       = 2,
    D3DZB_FORCE_DWORD = 0x7fffffff,
} D3DZBUFFERTYPE;
```

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

The D3DZB_FALSE and D3DZB_TRUE values are interchangeable with the TRUE and FALSE macro values previously used with D3DRENDERSTATE_ZENABLE.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice7::SetRenderState, Depth Buffers

Obsolete Enumerated Types

The following enumerated types are obsolete. For information about these enumerated types, see the documentation provided with a previous release of DirectX. Legacy documentation is provided with this SDK and is available for download from <http://www.microsoft.com/directx>.

- **D3DOPCODE**
- **D3DLIGHTSTATETYPE**
- **D3DTEXTUREBLEND**
- **D3DTEXTUREFILTER**

Obsolete Render States

The **D3DRENDERSTATETYPE** enumerated type, declared in D3dtypes.h, includes many render states that became obsolete with the release of DirectX 7.0. These render states are not supported by the DirectX 7.0 interfaces and are only declared for backward compatibility with existing applications. Attempts to use these render states with the **IDirect3DDevice7** interface will fail, returning **DDERR_INVALIDPARAMS**. Information about using these render states with legacy interfaces is provided with previous releases of the DirectX SDK. Legacy documentation is provided with this SDK and is available for download from <http://www.microsoft.com/directx>.

- **D3DRENDERSTATE_ANISOTROPY**
- **D3DRENDERSTATE_BORDERCOLOR**
- **D3DRENDERSTATE_FLUSHBATCH**

- D3DRENDERSTATE_MIPMAPLODBIAS
- D3DRENDERSTATE_MONOENABLE
- D3DRENDERSTATE_PLANEMASK
- D3DRENDERSTATE_ROP2
- D3DRENDERSTATE_STIPPLEENABLE
- D3DRENDERSTATE_STIPPLEPATTERN00 through
D3DRENDERSTATE_STIPPLEPATTERN31
- D3DRENDERSTATE_SUBPIXEL
- D3DRENDERSTATE_SUBPIXELX
- D3DRENDERSTATE_TEXTUREADDRESS
- D3DRENDERSTATE_TEXTUREADDRESSU
- D3DRENDERSTATE_TEXTUREADDRESSV
- D3DRENDERSTATE_TEXTUREHANDLE
- D3DRENDERSTATE_TEXTUREMAG
- D3DRENDERSTATE_TEXTUREMAPBLEND
- D3DRENDERSTATE_TEXTUREMIN
- D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT
- D3DRENDERSTATE_WRAPU and
- D3DRENDERSTATE_WRAPV

Other Types

This section contains information about the following Direct3D Immediate Mode types that are neither structures nor enumerated types:

- **D3DCOLOR**
- **D3DCOLORMODEL**
- **D3DFIXED**
- **D3DVALUE**

D3DCOLOR

The **D3DCOLOR** type is the fundamental Direct3D color type.

```
typedef DWORD D3DCOLOR, D3DCOLOR, *LPD3DCOLOR;
```

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DRGB, D3DRGBA

D3DCOLORMODEL

The **D3DCOLORMODEL** type is used to define the color model in which the system runs. A driver can expose either or both flags in the **dcmColorModel** member of the **D3DDEVICEDESC7** structure.

```
typedef DWORD D3DCOLORMODEL
```

D3DCOLOR_MONO

Use a monochromatic model (or ramp model). In this model, the blue component of a vertex color is used to define the brightness of a lit vertex.

D3DCOLOR_RGB

Use a full RGB model.

Remarks

Prior to DirectX 5.0, these values were part of an enumerated type. The enumerated type in earlier versions of DirectX had this syntax:

```
typedef enum _D3DCOLORMODEL {  
    D3DCOLOR_MONO = 1,  
    D3DCOLOR_RGB = 2,  
} D3DCOLORMODEL;
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 6.0 or later.

Header: Declared in d3dtypes.h.

See Also

D3DDEVICEDESC7

D3DFIXED

The **D3DFIXED** type is used to represent a 16:16 fixed-point value.

```
typedef LONG D3DFIXED;
```

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 5.0 or later.

Header: Declared in d3dtypes.h.

D3DVALUE

The **D3DVALUE** type is the fundamental Direct3D fractional data type.

```
typedef float D3DVALUE, *LPD3DVALUE;
```

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Windows CE: Unsupported.

Version: Requires DirectX 2.0 or later.

Header: Declared in d3dtypes.h.

Flexible Vertex Format Flags

Direct3D Immediate Mode uses flag values to describe vertex formats used for DrawPrimitive-based rendering. The D3dtypes.h header file declares the following flags to explicitly describe a vertex format and provides helper macros that act as common combinations of such flags. For more information, 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_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. If you use this flag, you must also specify a vertex normal, a vertex color component (D3DFVF_DIFFUSE or D3DFVF_SPECULAR), or include at least one set of texture coordinates (D3DFVF_TEX1 through D3DFVF_TEX8).

D3DFVF_XYZRHW

Vertex format includes the position of a transformed vertex. This flag cannot be used with the D3DFVF_XYZ or D3DFVF_NORMAL flags. If you use this flag, you must also specify a vertex color component (D3DFVF_DIFFUSE or D3DFVF_SPECULAR) or include at least one set of texture coordinates (D3DFVF_TEX1 through D3DFVF_TEX8).

D3DFVF_XYZB1 through D3DFVF_XYZB5

Vertex format contains position data, and a corresponding number of weighting (beta) values to be used for multimatrix vertex blending operations. Currently, 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.

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.

Helper macros**D3DFVF_LVERTEX**

Vertex format is equivalent to the **D3DLVERTEX** vertex type.

D3DFVF_TLVERTEX

Vertex format is equivalent to the **D3DTLVERTEX** vertex type.

D3DFVF_VERTEX

Vertex format is equivalent to the **D3DVERTEX** vertex type.

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_RESERVED1,

This bit is reserved to indicate that the system should emulate **D3DLVERTEX** processing. If this flag is used, the D3DFVF_XYZ, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX1 flags must also be used. This equates to the effect of the **D3DFVF_LVERTEX** helper macro.

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. 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

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 **IDirect3DDevice7::SetTextureStageState** and

IDirect3DDevice7::GetTextureStageState methods, specifying the D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAARG1, or D3DTSS_ALPHAARG2 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 $s - 2$.)

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.

D3DTA_SELECTMASK

Mask value for all arguments; not used when setting texture arguments.

D3DTA_TEXTURE

The texture argument is the texture color for this texture stage. This is valid only for the first color and alpha arguments in a stage (the D3DTSS_COLORARG1 and D3DTSS_ALPHAARG1 members of **D3DTEXTURESTAGESTATETYPE**).

D3DTA_TFACTOR

The texture argument is the texture factor set in a previous call to the **IDirect3DDevice7::SetRenderState** with the D3DRENDERSTATE_TEXTUREFACTOR render-state value.

D3DTA_SPECULAR

The texture argument is the specular color interpolated from vertex components during Gouraud shading. If the vertex does not contain a diffuse color, the default color is 0xFFFFFFFF.

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 minus x .

Return Values

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all Direct3D Immediate Mode methods. See the individual method descriptions for lists of the values that each can return.

D3D_OK

No error occurred.

D3DERR_BADMAJORVERSION

The service that you requested is unavailable in this major version of DirectX. (A major version denotes a primary release, such as DirectX 6.0.)

D3DERR_BADMINORVERSION

The service that you requested is available in this major version of DirectX, but not in this minor version. Get the latest version of the component run time from Microsoft. (A minor version denotes a secondary release, such as DirectX 6.1.)

D3DERR_COLORKEYATTACHED

The application attempted to create a texture with a surface that uses a color key for transparency.

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 palettized textures simultaneously enabled also share the same palette.

D3DERR_CONFLICTINGRENDERSTATE

The currently set render states cannot be used together.

D3DERR_DEVICEAGGREGATED

The **IDirect3DDevice7::SetRenderTarget** method was called on a device that was retrieved from the render target surface.

D3DERR_EXECUTE_CLIPPED_FAILED

The execute buffer could not be clipped during execution.

D3DERR_EXECUTE_CREATE_FAILED

The execute buffer could not be created. This typically occurs when no memory is available to allocate the execute buffer.

D3DERR_EXECUTE_DESTROY_FAILED

The memory for the execute buffer could not be deallocated.

D3DERR_EXECUTE_FAILED

The contents of the execute buffer are invalid and cannot be executed.

D3DERR_EXECUTE_LOCK_FAILED

The execute buffer could not be locked.

D3DERR_EXECUTE_LOCKED

The operation requested by the application could not be completed because the execute buffer is locked.

D3DERR_EXECUTE_NOT_LOCKED

The execute buffer could not be unlocked because it is not currently locked.

D3DERR_EXECUTE_UNLOCK_FAILED

The execute buffer could not be unlocked.

D3DERR_INBEGIN

The requested operation cannot be completed while scene rendering is taking place. Try again after the scene is completed and the **IDirect3DDevice7::EndScene** method is called.

D3DERR_INBEGINSTATEBLOCK

The operation cannot be completed while recording states for a state block. Complete recording by calling the **IDirect3DDevice7::EndStateBlock** method, and try again.

D3DERR_INITFAILED

A rendering device could not be created because the new device could not be initialized.

D3DERR_INVALID_DEVICE

The requested device type is not valid.

D3DERR_INVALIDCURRENTVIEWPORT

The currently selected viewport is not valid.

D3DERR_INVALIDMATRIX

The requested operation could not be completed because the combination of the currently set world, view, and projection matrices is invalid (the determinant of the combined matrix is 0).

D3DERR_INVALIDPALETTE

The palette associated with a surface is invalid.

D3DERR_INVALIDPRIMITIVETYPE

The primitive type specified by the application is invalid.

D3DERR_INVALIDRAMPTEXTURE

Ramp mode is being used, and the texture handle in the current material does not match the current texture handle that is set as a render state.

D3DERR_INVALIDSTATEBLOCK

The state block handle is invalid.

D3DERR_INVALIDVERTEXFORMAT

The combination of flexible vertex format flags specified by the application is not valid.

D3DERR_INVALIDVERTEXTYPE

The vertex type specified by the application is invalid.

D3DERR_LIGHT_SET_FAILED

The attempt to set lighting parameters for a light object failed.

D3DERR_LIGHTHASVIEWPORT

The requested operation failed because the light object is associated with another viewport.

D3DERR_LIGHTNOTINTHISVIEWPORT

The requested operation failed because the light object has not been associated with this viewport.

D3DERR_MATERIAL_CREATE_FAILED

The material could not be created. This typically occurs when no memory is available to allocate for the material.

D3DERR_MATERIAL_DESTROY_FAILED

The memory for the material could not be deallocated.

D3DERR_MATERIAL_GETDATA_FAILED

The material parameters could not be retrieved.

D3DERR_MATERIAL_SETDATA_FAILED

The material parameters could not be set.

D3DERR_MATRIX_CREATE_FAILED

The matrix could not be created. This can occur when no memory is available to allocate for the matrix.

D3DERR_MATRIX_DESTROY_FAILED

The memory for the matrix could not be deallocated.

D3DERR_MATRIX_GETDATA_FAILED

The matrix data could not be retrieved. This can occur when the matrix was not created by the current device.

D3DERR_MATRIX_SETDATA_FAILED

The matrix data could not be set. This can occur when the matrix was not created by the current device.

D3DERR_NOCURRENTVIEWPORT

The viewport parameters could not be retrieved because none have been set.

D3DERR_NOTINBEGIN

The requested rendering operation could not be completed because scene rendering has not begun. Call **IDirect3DDevice7::BeginScene** to begin rendering, and try again.

D3DERR_NOTINBEGINSTATEBLOCK

The requested operation could not be completed because it is only valid while recording a state block. Call the **IDirect3DDevice7::BeginStateBlock** method, and try again.

D3DERR_NOVIEWPORTS

The requested operation failed because the device currently has no viewports associated with it.

D3DERR_SCENE_BEGIN_FAILED

Scene rendering could not begin.

D3DERR_SCENE_END_FAILED

Scene rendering could not be completed.

D3DERR_SCENE_IN_SCENE

Scene rendering could not begin because a previous scene was not completed by a call to the **IDirect3DDevice7::EndScene** method.

D3DERR_SCENE_NOT_IN_SCENE

Scene rendering could not be completed because a scene was not started by a previous call to the **IDirect3DDevice7::BeginScene** method.

D3DERR_SETVIEWPORTDATA_FAILED

The viewport parameters could not be set.

D3DERR_STENCILBUFFER_NOTPRESENT

The requested stencil buffer operation could not be completed because there is no stencil buffer attached to the render target surface.

D3DERR_SURFACENOTINVIDMEM

The device could not be created because the render target surface is not located in video memory. (Hardware-accelerated devices require video-memory render target surfaces.)

D3DERR_TEXTURE_BADSIZE

The dimensions of a current texture are invalid. This can occur when an application attempts to use a texture that has dimensions that are not a power of 2 with a device that requires them.

D3DERR_TEXTURE_CREATE_FAILED

The texture handle for the texture could not be retrieved from the driver.

D3DERR_TEXTURE_DESTROY_FAILED

The device was unable to deallocate the texture memory.

D3DERR_TEXTURE_GETSURF_FAILED

The DirectDraw surface used to create the texture could not be retrieved.

D3DERR_TEXTURE_LOAD_FAILED

The texture could not be loaded.

D3DERR_TEXTURE_LOCK_FAILED

The texture could not be locked.

D3DERR_TEXTURE_LOCKED

The requested operation could not be completed because the texture surface is currently locked.

D3DERR_TEXTURE_NO_SUPPORT

The device does not support texture mapping.

D3DERR_TEXTURE_NOT_LOCKED

The requested operation could not be completed because the texture surface is not locked.

D3DERR_TEXTURE_SWAP_FAILED

The texture handles could not be swapped.

D3DERR_TEXTURE_UNLOCK_FAILED

The texture surface could not be unlocked.

D3DERR_TOOMANYOPERATIONS

The application is requesting more texture-filtering operations than the device supports.

D3DERR_TOOMANYPRIMITIVES

The device is unable to render the provided number of primitives in a single pass.

D3DERR_UNSUPPORTEDALPHAARG

The device does not support one of the specified texture-blending arguments for the alpha channel.

D3DERR_UNSUPPORTEDALPHAOPERATION

The device does not support one of the specified texture-blending operations for the alpha channel.

D3DERR_UNSUPPORTEDCOLORARG

The device does not support one of the specified texture-blending arguments for color values.

D3DERR_UNSUPPORTEDCOLOROPERATION

The device does not support one of the specified texture-blending operations 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_VBUF_CREATE_FAILED

The vertex buffer could not be created. This can happen when there is insufficient memory to allocate a vertex buffer.

D3DERR_VERTEXBUFFERLOCKED

The requested operation could not be completed because the vertex buffer is locked.

D3DERR_VERTEXBUFFEROPTIMIZED

The requested operation could not be completed because the vertex buffer is optimized. (The contents of optimized vertex buffers are driver-specific and considered private.)

D3DERR_VERTEXBUFFERUNLOCKFAILED

The vertex buffer could not be unlocked because the vertex buffer memory was overrun. Be sure that your application does not write beyond the size of the vertex buffer.

D3DERR_VIEWPORTDATANOTSET

The requested operation could not be completed because viewport parameters have not yet been set. Set the viewport parameters by calling the **IDirect3DDevice7::SetViewport** method, and try again.

D3DERR_VIEWPORTHASNODEVICE

This value is used only by the **IDirect3DDevice3** interface and its predecessors. For the **IDirect3DDevice7** interface, this error value is not used.

The requested operation could not be completed because the viewport has not yet been associated with a device. Associate the viewport with a rendering device by calling the **IDirect3DDevice3::AddViewport** method, and try again.

D3DERR_WRONGTEXTUREFORMAT

The pixel format of the texture surface is not valid.

D3DERR_ZBUFF_NEEDS_SYSTEMMEMORY

The requested operation could not be completed because the specified device requires system-memory depth-buffer surfaces. (Software rendering devices require system-memory depth buffers.)

D3DERR_ZBUFF_NEEDS_VIDEOMEMORY

The requested operation could not be completed because the specified device requires video-memory depth-buffer surfaces. (Hardware-accelerated devices require video-memory depth buffers.)

D3DERR_ZBUFFER_NOTPRESENT

The requested operation could not be completed because the render target surface does not have an attached depth buffer.

Direct3D Immediate Mode Visual Basic Reference

This section contains reference information for the application programming interface (API) elements provided by Direct3D® Immediate Mode. Reference material is divided into the following categories:

- Classes
- Types
- Enumerations
- Flexible Vertex Format Flags
- Texture Argument Flags
- Error Codes

Classes

This section contains reference information for the classes provided by Direct3D Immediate Mode. The following classes are covered:

- **Direct3D7**
- **Direct3DDevice7**
- **Direct3DEnumDevices**
- **Direct3DEnumPixelFormat**
- **Direct3DVertexBuffer7**

Direct3D7

Applications use the methods of the **Direct3D7** class to create Direct3D objects and set up the environment. Moreover, The **Direct3D7** class enables applications to create vertex buffers and enumerate texture map and depth-buffer formats. This section is a reference to the methods of this class.

The **Direct3D7** class is obtained by calling the **GetDirect3D** method from a **DirectDraw7** object.

The methods of the **Direct3D7** class can be organized into the following groups:

Creation	CreateDevice
	CreateVertexBuffer
Enumeration	GetDevicesEnum
	GetEnumZBufferFormats
Miscellaneous	EvictManagedTextures
	GetDirectDraw

See Also

Accessing Direct3D, Direct3D and DirectDraw

Direct3D7.CreateDevice

The **Direct3D7.CreateDevice** method creates a Direct3D device to be used with the DrawPrimitive methods.

```
object.CreateDevice( _  
    guid As String, _  
    surf As DirectDrawSurface7) As Direct3DDevice7
```

Parameters

object

Object expression that resolves to a **Direct3D7** object.

guid

Guid for the new device. This value can be an enumerated GUID string or the "IID_Direct3DHALDevice" or "IID_Direct3DRGBDevice" string constants. The method also accepts "IID_Direct3DRefDevice" which creates the reference device, for use in testing or feature demonstration.

surf

IDH_Direct3D7_d3d_vb

IDH_Direct3D7.CreateDevice_d3d_vb

A **DirectDrawSurface7** object for the DirectDrawSurface object that will be the device's rendering target. The surface must have been created as a 3-D device by using the DDSCAPS_3DDEVICE capability.

Return Values

If the method succeeds, the return value is a reference to a **Direct3DDevice7** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the parameters is invalid.

Remarks

All rendering devices created by a given **Direct3D7** object share the same physical resources. Although your application can create multiple rendering devices from a single **Direct3D7** object, because they share the same hardware, extreme performance penalties will be incurred.

When you call **Direct3D7.CreateDevice**, you create a device object that is separate from a DirectDraw surface object. This device uses a DirectDraw surface as a rendering target.

See Also

Direct3DDevice7, Creating a Direct3D Device, Direct3D Devices

Direct3D7.CreateVertexBuffer

The **Direct3D7.CreateVertexBuffer** method creates a vertex buffer object.

```
object.CreateVertexBuffer( _  
    desc As D3DVERTEXBUFFERDESC, _  
    flags As CONST_D3DDPFLAGS) As Direct3DVertexBuffer7
```

Parameters

object

Object expression that resolves to a **Direct3D7** object.

desc

A **D3DVERTEXBUFFERDESC** type that describes the format and number of vertices that the vertex buffer will contain.

flags

IDH_Direct3D7.CreateVertexBuffer_d3d_vb

One of the constants of the **CONST_D3DDPFLAGS** enumeration (flexible vertex format flags) representing the clipping value. Set this parameter to 0 to create a vertex buffer that can contain clipping information for untransformed or transformed vertices, or use the **D3DDP_DONOTCLIP** flag to create a vertex buffer that will contain transformed vertices, but no clipping information.

Return Values

If the method succeeds, a reference to a **Direct3DVertexBuffer7** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes:

- D3DERR_INVALIDVERTEXFORMAT
- D3DERR_VBUF_CREATE_FAILED
- DDERR_INVALIDOBJECT
- DDERR_INVALIDPARAMS
- DDERR_OUTOFMEMORY

See Also

Direct3DVertexBuffer7, Vertex Buffers

Direct3D7.EvictManagedTextures

The **Direct3D7.EvictManagedTextures** method purges all managed textures from local or non-local video memory.

object.**EvictManagedTextures()**

Parameters

object

Object expression that resolves to a **Direct3D7** object.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

IDH_Direct3D7.EvictManagedTextures_d3d_vb

Remarks

This method causes Direct3D to remove any texture surfaces created with the DDSCAPS2_TEXTUREMANAGE or DDSCAPS2_D3DTEXTUREMANAGE flags from local or non-local video memory.

See Also

Automatic Texture Management

Direct3D7.GetDevicesEnum

The **Direct3D7.GetDevicesEnum** method creates a **Direct3DEnumDevices** object.

object.GetDevicesEnum() As Direct3DEnumDevices

Parameters

object

Object expression that resolves to a **Direct3D7** object.

Return Values

If the method succeeds, a **Direct3DEnumDevices** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Direct3D7.GetDirectDraw

The **Direct3D7.GetDirectDraw** method retrieves the **DirectDraw7** object associated with this Direct3D object.

object.GetDirectDraw() As DirectDraw7

Parameters

object

Object expression that resolves to a **Direct3D7** object.

IDH_Direct3D7.GetDevicesEnum_d3d_vb

IDH_Direct3D7.GetDirectDraw_d3d_vb

Return Values

If the method succeeds, a reference to a **DirectDraw7** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

Direct3D7.GetEnumZBufferFormats

The **Direct3D7.GetEnumZBufferFormats** method creates a **Direct3DEnumPixelFormat** object.

object.**GetEnumZBufferFormats**(
 guid As String) As Direct3DEnumPixelFormat

Parameters

object

Object expression that resolves to a **Direct3D7** object.

guid

A globally unique identifier for the device whose depth-buffer formats will be enumerated.

Return Values

If the method succeeds, a reference to a **Direct3DEnumPixelFormat** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOZBUFFERHW
DDERR_OUTOFMEMORY

IDH_Direct3D7.GetEnumZBufferFormats_d3d_vb

For information on trapping errors, see the Visual Basic Error Trapping topic.

Direct3DDevice7

The **Direct3DDevice7** class provides methods enabling applications to perform DrawPrimitive-based rendering. You create a device by calling the **Direct3D7.CreateDevice** method.

For a conceptual overview, see Direct3D Devices and the DrawPrimitive Methods.

The methods of the **Direct3DDevice7** class can be organized into the following groups:

Information

GetCaps

GetDirect3D

GetInfo

GetDeviceGuid

Device States

ApplyStateBlock

BeginStateBlock

CaptureStateBlock

CreateStateBlock

DeleteStateBlock

EndStateBlock

GetClipStatus

GetRenderState

GetRenderStateSingle

GetRenderTarget

GetTransform

SetClipStatus

SetRenderState

SetRenderStateSingle

SetRenderTarget

SetTransform

Lighting and Materials

GetMaterial

GetLight

GetLightEnable

LightEnable

SetMaterial

	SetLight
Miscellaneous	ComputeSphereVisibility
	MultiplyTransform
Rendering	DrawIndexedPrimitive
	DrawIndexedPrimitiveVB
	DrawPrimitive
	DrawPrimitiveVB
Scenes	BeginScene
	EndScene
Textures	GetTexture
	GetTextureFormatsEnum
	GetTextureStageState
	GetTextureStageStateSingle
	Load
	PreLoad
	SetTexture
	SetTextureStageState
	SetTextureStageStateSingle
	ValidateDevice
User-defined clip planes	GetClipPlane
	SetClipPlane
Viewports	Clear
	GetViewport
	SetViewport

This class contains methods to support more flexible vertex formats, vertex buffers, and visibility computation.

See Also

Direct3D Devices, Rendering

Direct3DDevice7.ApplyStateBlock

The **Direct3DDevice7.ApplyStateBlock** method applies an existing device-state block to the rendering device.

```
object.ApplyStateBlock( _  
    blockHandle As Long)
```

IDH_Direct3DDevice7.ApplyStateBlock_d3d_vb

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

blockHandle

Handle to the device state block to be applied, as returned by a previous call to the **Direct3DDevice7.EndStateBlock** method.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to **DDERR_INVALIDPARAMS** to indicate that the *blockHandle* parameter is invalid or a macro is currently being recorded.

Remarks

Applications cannot apply a device state block while recording another block.

This method was introduced with the **Direct3DDevice7** interface.

As with all operations that affect the state of the rendering device, it is recommend that you apply state blocks during scene rendering—that is, after calling the **Direct3DDevice7.BeginScene** method, and before calling **Direct3DDevice7.EndScene**.

See Also

Direct3DDevice7.BeginStateBlock, **Direct3DDevice7.CaptureStateBlock**, **Direct3DDevice7.CreateStateBlock**, **Direct3DDevice7.DeleteStateBlock**, **Direct3DDevice7.EndStateBlock**, State Blocks

Direct3DDevice7.BeginScene

The **Direct3DDevice7.BeginScene** method begins a scene. Applications must call this method before performing any rendering, and must call **Direct3DDevice7.EndScene** when rendering is complete, and before calling **Direct3DDevice7.BeginScene** again.

object.**BeginScene()**

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

IDH_Direct3DDevice7.BeginScene_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

If the **BeginScene** method fails, the device was unable to begin the scene, and there is no need to call the **Direct3DDevice7.EndScene** method. In fact, calls to the **Direct3DDevice7.EndScene** method will fail if the previous call to **BeginScene** failed.

See Also

Direct3DDevice7.EndScene

Direct3DDevice7.BeginStateBlock

The **Direct3DDevice7.BeginStateBlock** method signals Direct3D to begin recording a device state block.

object.**BeginStateBlock()**

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes:

DDERR_INVALIDPARAMS

DDERR_OUTOFMEMORY

Remarks

Applications can ensure that all recorded states are valid by calling the **Direct3DDevice7.ValidateDevice** method prior to calling this method.

The following methods can be recorded in a state block (that is, after calling **Direct3DDevice7.BeginStateBlock** and before **Direct3DDevice7.EndStateBlock**):

- **Direct3DDevice7.LightEnable**
- **Direct3DDevice7.SetClipPlane**
- **Direct3DDevice7.SetLight**

IDH_Direct3DDevice7.BeginStateBlock_d3d_vb

- **Direct3DDevice7.SetMaterial**
- **Direct3DDevice7.SetRenderState**
- **Direct3DDevice7.SetTexture**
- **Direct3DDevice7.SetTextureStageState**
- **Direct3DDevice7.SetTransform**
- **Direct3DDevice7.SetViewport**

The ordering of state changes in a state block is not guaranteed. If the same state is specified multiple times in a state set only the last value will be used.

See Also

Direct3DDevice7.EndStateBlock, **Direct3DDevice7.CaptureStateBlock**, **Direct3DDevice7.CreateStateBlock**, **Direct3DDevice7.DeleteStateBlock**, **Direct3DDevice7.ApplyStateBlock**, State Blocks

Direct3DDevice7.CaptureStateBlock

The **Direct3DDevice7.CaptureStateBlock** method updates the values within an existing state block to the values currently set for the device.

*object.CaptureStateBlock(_
 blockHandle As Long)*

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

blockHandle

Handle to the state block into which the current device state will be captured.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes:

D3DERR_INBEGINSTATEBLOCK
D3DERR_INOVERLAYSTATEBLOCK

Remarks

This method captures updated values for states within an existing state block. It does not capture the entire state of the device. For more information, see [Capturing State Blocks](#).

See Also

Direct3DDevice7.ApplyStateBlock, **Direct3DDevice7.BeginStateBlock**, **Direct3DDevice7.CreateStateBlock**, **Direct3DDevice7.EndStateBlock**, **Direct3DDevice7.DeleteStateBlock**, [State Blocks](#)

Direct3DDevice7.CreateStateBlock

The **Direct3DDevice7.CreateStateBlock** method creates a new state block that contains the current values for all device states, vertex-related states, or pixel-related states.

```
object.CreateStateBlock( _  
    d3dsbType As CONST_D3DSTATEBLOCKTYPE, _  
    blockHandle As Long)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

d3dsbType

Type of state data that the method should capture. This can be one of the values defined in the **CONST_D3DSTATEBLOCKTYPE** enumeration.

blockHandle

Variable that will contain the state block handle if the method succeeds.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes:

```
D3DERR_INBEGINSTATEBLOCK  
DDERR_INVALIDPARAMS  
DDERR_OUTOFMEMORY
```

Remarks

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

IDH_Direct3DDevice7.CreateStateBlock_d3d_vb

the system processes pixel or depth-buffer data during rasterization. Some states are contained in both groups. For information about the states in each group, see Creating Predefined State Blocks.

See Also

Direct3DDevice7.ApplyStateBlock, **Direct3DDevice7.BeginStateBlock**, **Direct3DDevice7.CaptureStateBlock**, **Direct3DDevice7.EndStateBlock**, **Direct3DDevice7.DeleteStateBlock**, State Blocks

Direct3DDevice7.Clear

The **Direct3DDevice7.Clear** method 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, _
    recs() As D3DRECT, _
    flags As CONST_D3DCLEARFLAGS, _
    color As Long, _
    z As Single, _
    stencil As Long)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

count

Number of rectangles in the array at *recs*.

recs()

Array of **D3DRECT** types that describe the rectangles to be cleared. 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.

flags

Flags indicating which surfaces should be cleared. This parameter can be any combination of the following flags, but at least one flag must be used:

D3DCLEAR_TARGET

Clear the rendering target to the color in the *color* parameter.

D3DCLEAR_STENCIL

Clear the stencil buffer to the value in the *stencil* parameter.

D3DCLEAR_ZBUFFER

Clear the depth-buffer to the value in the *z* parameter.

IDH_Direct3DDevice7.Clear_d3d_vb

color

RGBA color value to which the render target surface will be cleared. You can construct an appropriate value by calling the **DirectX7.CreateColorRGBA** method.

z

New z-value that this method stores in the depth-buffer. This parameter can range from 0.0 to 1.0, inclusive (for z-based or w-based depth buffers). The 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 range from 0 to 2^n-1 inclusive, where n is the bit depth of the stencil buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes:

D3DERR_STENCILBUFFER_NOTPRESENT
D3DERR_VIEWPORTHASNODEVICE
D3DERR_ZBUFFER_NOTPRESENT
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

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 doesn't contain stencil buffer information, this method fails.

Direct3DDevice7.ComputeSphereVisibility

The **Direct3DDevice7.ComputeSphereVisibility** method calculates the visibility (complete, partial, or no visibility) of a sphere within the current viewport for this device.

object.ComputeSphereVisibility(_
 center As D3DVECTOR, _
 radius As Single) As Long

IDH_Direct3DDevice7.ComputeSphereVisibility_d3d_vb

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

center

A **D3DVECTOR** type describing the center point for the sphere, in world-space coordinates.

radius

The radius for the sphere.

Return Values

If the method succeeds, the return value is a combination of flags from the **CONST_D3DCLIPFLAGS** enumeration that describe the visibility of that sphere within the current viewport for this device. If the return value is zero (some combination of the "inside" flags) the sphere is completely visible.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Sphere visibility is computed by back transforming the viewing frustum to the model space, using the inverse of the combined world, view or projection matrices. If the combined matrix cannot be inverted (if the determinant is zero), the method fails, returning DDERR_GENERIC.

Direct3DDevice7.DeleteStateBlock

The **Direct3DDevice7.DeleteStateBlock** method deletes a previously recorded device state block.

*object.DeleteStateBlock(_
 blockHandle As Long)*

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

blockHandle

Handle to the device state block to be deleted, as returned by a previous call to the **Direct3DDevice7.EndStateBlock** method.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS to indicate that the *blockHandle* parameter is invalid or a macro is currently being recorded.

Remarks

Applications cannot delete a device state block while another is being recorded.

See Also

Direct3DDevice7.BeginStateBlock, **Direct3DDevice7.CaptureStateBlock**, **Direct3DDevice7.CreateStateBlock**, **Direct3DDevice7.EndStateBlock**, **Direct3DDevice7.ApplyStateBlock**, State Blocks

Direct3DDevice7.DrawIndexedPrimitive

The **Direct3DDevice7.DrawIndexedPrimitive** method renders the specified geometric primitive based on indexing into an array of vertices.

```
object.DrawIndexedPrimitive( _  
    d3Dpt As CONST_D3DPRIMITIVETYPE, _  
    d3dfvf As CONST_D3DFVFFLAGS, _  
    vertices As Any, _  
    vertexCount As Long, _  
    indices() As Integer, _  
    IndicesCount As Long, _  
    flags As CONST_D3DDPFLAGS)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

d3Dpt

Type of primitive to be rendered by this command. This must be one of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration.

IDH_Direct3DDevice7.DrawIndexedPrimitive_d3d_vb

Note that the D3DPT_POINTLIST member of **CONST_D3DPRIMITIVETYPE** is not indexed.

d3dfvf

A combination of flexible vertex format flags from the **CONST_D3DFVFFLAGS** enumeration that describe the vertex format used.

vertices

First element in an array of vertices to be used in the primitive sequence.

vertexCount

Defines the number of vertices in the *vertices* array.

Notice that this parameter is used differently from the *vertexCount* parameter in the **Direct3DDevice7.DrawPrimitive** method. In that method, the *vertexCount* parameter gives the number of vertices to draw, but here it gives the total number of vertices in the array pointed to by the *vertices* parameter. When you call **Direct3DDevice7.DrawIndexedPrimitive**, you specify the number of vertices to draw in the *IndicesCount* parameter.

indices()

Array that is to be used to index into the specified vertex list when creating the geometry to render. The values in the array must index vertices within the range [0, *dwNumVertices* - 1].

IndicesCount

Specifies the number of indices provided for creating the geometry. The maximum number of indices allowed is 65535 (&HFFFF).

flags

One of the constants from the **CONST_D3DDPFLAGS** defining how the primitive is drawn.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE
 D3DERR_INVALIDPRIMITIVETYPE
 D3DERR_INVALIDVERTEXTYPE
 DDERR_WASSTILLDRAWING
 DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

In current versions of DirectX, **Direct3DDevice7.DrawIndexedPrimitive** can sometimes generate an update rectangle that is larger than it strictly needs to be. If a large number of vertices need to be processed, this can have a negative impact on the performance of your application. If you are using **D3DTLVERTEX** vertices and the system is processing more vertices than you need, you should use the **D3DDP_DONOTCLIP** and **D3DDP_DONOTUPDATEEXTENTS** flags to solve the problem.

See Also

Direct3DDevice7.DrawPrimitive, **Direct3DDevice7.DrawPrimitiveVB**,
Direct3DDevice7.DrawIndexedPrimitiveVB

Direct3DDevice7.DrawIndexedPrimitiveVB

The **Direct3DDevice7.DrawIndexedPrimitiveVB** method renders a geometric primitive based on indexing into an array of vertices within a vertex buffer.

```
object.DrawIndexedPrimitiveVB( _  
    d3Dpt As CONST_D3DPRIMITIVETYPE, _  
    vertexBuffer As Direct3DVertexBuffer7, _  
    StartVertex As Long, _  
    NumVertices As Long, _  
    indexArray() As Integer, _  
    indexcount As Long, _  
    flags As CONST_D3DDPFLAGS)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

d3Dpt

Type of primitive to be rendered by this command. This must be one of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration.

Note that the **D3DPT_POINTLIST** member of **CONST_D3DPRIMITIVETYPE** is not indexed.

vertexBuffer

A **Direct3DVertexBuffer7** object for the vertex buffer that contains the array of vertices. Vertices can be transformed or untransformed, optimized or unoptimized.

StartVertex

Index of the first vertex in the vertex buffer that will be rendered.

NumVertices

IDH_Direct3DDevice7.DrawIndexedPrimitiveVB_d3d_vb

Total number of vertices in the vertex buffer that will be rendered.

indexArray()

An array that will be used to index into the vertices in the vertex buffer. The values in the array must index vertices within the range [*StartVertex*, *StartVertex* + *NumVertices* - 1].

indexcount

The number of indices in the *indexArray()* array. The maximum number of indices allowed is 65535 (&HFFFF).

flags

One of the constants of the **CONST_D3DDPFLAGS** enumeration defining how the primitive is drawn.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDPRIMITIVETYPE
 D3DERR_INVALIDTEXTTYPE
 D3DERR_VERTEXBUFFERLOCKED
 DDERR_INVALIDOBJECT
 DDERR_INVALIDPARAMS
 DDERR_WASSTILLDRAWING

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

With the exception of the reference rasterizer, software devices cannot render from a video memory (local or non-local) vertex buffer. To render a vertex buffer using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

You cannot render from a locked vertex buffer; calls to the **Direct3DDevice7.DrawIndexedPrimitiveVB** or **Direct3DDevice7.DrawPrimitiveVB** method using a locked buffer will fail, returning D3DERR_VERTEXBUFFERLOCKED.

See Also

Direct3DDevice7.DrawPrimitive, **Direct3DDevice7.DrawPrimitiveVB**,
Direct3DDevice7.DrawIndexedPrimitive

Direct3DDevice7.DrawPrimitive

The **Direct3DDevice7.DrawPrimitive** method renders the specified array of vertices as a sequence of geometric primitives of the specified type.

```
object.DrawPrimitive( _  
    d3Dpt As CONST_D3DPRIMITIVETYPE, _  
    d3dfvf As CONST_D3DFVFFLAGS, _  
    vertices As Any, _  
    vertexCount As Long, _  
    flags As CONST_D3DDPFLAGS)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

d3Dpt

Type of primitive to be rendered by this command. This must be one of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration.

Note that the D3DPT_POINTLIST member of **CONST_D3DPRIMITIVETYPE** is not indexed.

d3dfvf

A combination of flexible vertex format flags from the **CONST_D3DFVFFLAGS** enumeration that describe the vertex format used.

vertices

First element in an array of vertices to be used in the primitive sequence.

vertexCount

Defines the number of vertices in the vertices array. The maximum number of vertices allowed is 65535 (&HFFFF).

flags

One of the constants of the **CONST_D3DDPFLAGS** enumeration defining how the primitive is drawn.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE

IDH_Direct3DDevice7.DrawPrimitive_d3d_vb

D3DERR_INVALIDPRIMITIVETYPE
 D3DERR_INVALIDTEXTTYPE
 DDERR_WASSTILLDRAWING
 DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

See Also

Direct3DDevice7.DrawPrimitiveVB, **Direct3DDevice7.DrawIndexedPrimitive**, **Direct3DDevice7.DrawIndexedPrimitiveVB**

Direct3DDevice7.DrawPrimitiveVB

The **Direct3DDevice7.DrawPrimitiveVB** method renders an array of vertices in a vertex buffer as a sequence of geometric primitives.

```
object.DrawPrimitiveVB( _  
    d3dpt As CONST_D3DPRIMITIVETYPE, _  
    vertexBuffer As Direct3DVertexBuffer7, _  
    startVertex As Long, _  
    numVertices As Long, _  
    flags As CONST_D3DDPFLAGS)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

d3dpt

Type of primitive to be rendered by this command. This must be one of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration.

Note that the D3DPT_POINTLIST member of **CONST_D3DPRIMITIVETYPE** is not indexed.

vertexBuffer

A **Direct3DVertexBuffer7** object for the vertex buffer that contains the array of vertices. Vertices can be transformed or untransformed, optimized or unoptimized.

IDH_Direct3DDevice7.DrawPrimitiveVB_d3d_vb

startVertex

Index value of the first vertex in the primitive. The highest possible starting index is 65535 (&HFFFF).

numVertices

Number of vertices to be rendered. The maximum number of vertices allowed is 65535 (&HFFFF).

flags

One of the constants of the **CONST_D3DDPFLAGS** enumeration defining how the primitive is drawn.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE
D3DERR_INVALIDPRIMITIVETYPE
D3DERR_INVALIDVERTEXTYPE
D3DERR_VERTEXBUFFERLOCKED
DDERR_WASSTILLDRAWING
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

With the exception of the reference rasterizer, software devices cannot render from a video memory (local or non-local) vertex buffer. To render a vertex buffer using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

You cannot render from a locked vertex buffer; calls to the **Direct3DDevice7.DrawIndexedPrimitiveVB** or **Direct3DDevice7.DrawPrimitiveVB** method using a locked buffer will fail, returning D3DERR_VERTEXBUFFERLOCKED.

See Also

Direct3DDevice7.DrawPrimitive, **Direct3DDevice7.DrawIndexedPrimitive**, **Direct3DDevice7.DrawIndexedPrimitiveVB**

Direct3DDevice7.EndScene

The **Direct3DDevice7.EndScene** method ends a scene that was begun by calling the **Direct3DDevice7.BeginScene** method.

object.EndScene()

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

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 **Direct3DDevice7.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

Direct3DDevice7.BeginScene

Direct3DDevice7.EndStateBlock

The **Direct3DDevice7.EndStateBlock** method signals Direct3D to stop recording a device state block and retrieve the block handle.

object.EndStateBlock(_
 blockHandle **As Long**)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

blockHandle

IDH_Direct3DDevice7.EndScene_d3d_vb

IDH_Direct3DDevice7.EndStateBlock_d3d_vb

Variable that will be filled with the handle to the completed device state block. This value is used with the **Direct3DDevice7.ApplyStateBlock** and **Direct3DDevice7.DeleteStateBlock** methods.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to **DDERR_INVALIDPARAMS** to indicate that the *blockHandle* parameter is invalid or that the **Direct3DDevice7.BeginStateBlock** method has not been called.

See Also

Direct3DDevice7.ApplyStateBlock, **Direct3DDevice7.BeginStateBlock**, **Direct3DDevice7.CaptureStateBlock**, **Direct3DDevice7.CreateStateBlock**, **Direct3DDevice7.DeleteStateBlock**, State Blocks

Direct3DDevice7.GetCaps

The **Direct3DDevice7.GetCaps** method retrieves the capabilities of the Direct3D device.

```
object.GetCaps( _  
    desc As D3DDEVICEDESC7)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

desc

A **D3DDEVICEDESC7** type that will contain the hardware features of the device.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

```
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS
```

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method does not retrieve the capabilities of the display device. To retrieve this information, use the **DirectDraw7.GetCaps** method.

Direct3DDevice7.GetClipPlane

The **Direct3DDevice7.GetClipPlane** method retrieves the coefficients of a user-defined clipping plane for the device.

```
object.GetClipPlane( _  
    Index As Long, _  
    a As Single, _  
    b As Single, _  
    c As Single, _  
    d As Single)
```

Parameters

Index

Index of the clipping plane for which the plane equation coefficients are retrieved.

a, b, c, and d

Values that represent the coefficients of the clipping plane, 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 DDERR_INVALIDPARAMS. This error indicates that the value in *Index* exceeds the maximum clipping plane index supported by the device.

Remarks

The coefficients that this method reports take the form of the general plane equation. The values in *a*, *b*, *c*, and *d* would 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 $Ax + By + Cz + Dw \geq 0$. 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 **Direct3DDevice7.SetClipPlane** method.

See Also

Direct3DDevice7.SetClipPlane, D3DRENDERSTATE_CLIPPLANEENABLE, User-defined Clip Planes

Direct3DDevice7.GetClipStatus

The **Direct3DDevice7.GetClipStatus** method gets the current clip status.

object.**GetClipStatus**(*clipStatus* As D3DCLIPSTATUS)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

clipStatus

A **D3DCLIPSTATUS** type that describes the current clip status.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the parameters is invalid.

See Also

Direct3DDevice7.SetClipStatus

Direct3DDevice7.GetDeviceGuid

The **Direct3DDevice7.GetDeviceGuid** method retrieves the globally unique identifier (GUID) for this device, as a **String**.

object.**GetDeviceGuid**() As String

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

Return Values

The method returns the GUID for this device, as type **String**.

IDH_Direct3DDevice7.GetClipStatus_d3d_vb

IDH_Direct3DDevice7.GetDeviceGuid_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

See Also

Direct3D Device Types

Direct3DDevice7.GetDirect3D

The **Direct3DDevice7.GetDirect3D** method retrieves the Direct3D object for this device.

object.**GetDirect3D()** As **Direct3D7**

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

Return Values

If the method succeeds, a reference to a **Direct3D7** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DDevice7.GetInfo

The **Direct3DDevice7.GetInfo** method retrieves information about the rendering device. Information can pertain to Direct3D or the underlying device driver.

object.**GetInfo**(_
 IDevInfoID As **CONST_D3DDEVINFOID**, _
 DevInfoType As **Any**, _
 ISize As **Long**)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

IDevInfoID

IDH_Direct3DDevice7.GetDirect3D_d3d_vb

IDH_Direct3DDevice7.GetInfo_d3d_vb

Flag from the **CONST_D3DDEVINFOID** enumeration that indicate the data type of the *DevInfoType* parameter.

DevInfoType

Variable that will contain the specified device information if the call succeeds. The data type of this variable, and how the data it contains is to be interpreted, is determined by the flag in *lDevInfoID*.

lSize

Size of the type at *DevInfoType*, in bytes, as returned by the **Len** Visual Basic function.

Return Values

None.

Error Codes

If the method fails, the return value may be **S_FALSE** to indicate that the device does not support information queries, or one of the following error codes:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method makes it possible for drivers to declare specific information types, and corresponding structures, that may not be documented in this SDK.

This method executes synchronously, and it can negatively impact your application's performance when it executes slowly. Do not call this method during scene rendering (between calls to **Direct3DDevice7.BeginScene** and **Direct3DDevice7.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 without retrieving device data. This occurs when the retail version of the DirectX runtime is installed on the host system.

See Also

D3DDEVINFO_TEXTUREMANAGER

Direct3DDevice7.GetLight

The **Direct3DDevice7.GetLight** method retrieves a set of lighting properties used by this device.

object.**GetLight**(_

IDH_Direct3DDevice7.GetLight_d3d_vb

*LightIndex As Long, _
Light As D3DLIGHT7)*

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

LightIndex

Zero-based index of the lighting property set to be retrieved.

Light

A **D3DLIGHT7** type that will be filled with the retrieved lighting-parameter set.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to **DDERR_INVALIDPARAMS** if one of the parameters is invalid.

See Also

Direct3DDevice7.SetLight, **Direct3DDevice7.GetLightEnable**,
Direct3DDevice7.LightEnable, Lighting and Materials

Direct3DDevice7.GetLightEnable

The **Direct3DDevice7.GetLightEnable** method retrieves the activity status—enabled or disabled—for a light.

*object.GetLightEnable(_
LightIndex As Long) As Boolean*

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

LightIndex

Zero-based index of the set of lighting parameters that are the target of this method.

Return Values

If the method succeeds, the return value indicates the activity status of the light. The method returns **True** to indicate that the specified light is enabled, and **False** to indicate that it is disabled.

IDH_Direct3DDevice7.GetLightEnable_d3d_vb

Error Codes

If the method fails, it sets **Err.Number** to an error code and raises an error. For a list of possible error codes, see Direct3D Immediate Mode Error Codes.

See Also

Direct3DDevice7.GetLight, **Direct3DDevice7.LightEnable**,
Direct3DDevice7.SetLight, Lighting and Materials

Direct3DDevice7.GetMaterial

The **Direct3DDevice7.GetMaterial** method retrieves the current material properties for the device.

object.**GetMaterial**(
 Material As **D3DMATERIAL7**)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

Material

A **D3DMATERIAL7** type that will be filled with the currently set material properties.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to **DDERR_INVALIDPARAMS** to indicate the *Material* parameter is invalid.

See Also

Direct3DDevice7.SetMaterial, Lighting and Materials

Direct3DDevice7.GetRenderState

The **Direct3DDevice7.GetRenderState** method gets a render-state value for a device.

object.**GetRenderState**(
 state As **CONST_D3DRENDERSTATETYPE**) As Long

IDH_Direct3DDevice7.GetMaterial_d3d_vb

IDH_Direct3DDevice7.GetRenderState_d3d_vb

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

state

Device state variable that is being queried. This parameter can be any of the constants of the **CONST_D3DRENDERSTATETYPE** enumeration.

Return Values

If the method succeeds, the return value is the current value of the specified render state.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to **DDERR_INVALIDPARAMS** if one of the parameters is invalid.

See Also

Direct3DDevice7.SetRenderState

Direct3DDevice7.GetRenderStateSingle

The **Direct3DDevice7.GetRenderStateSingle** method gets a render-state value for a device.

object.**GetRenderStateSingle**(
 state As **CONST_D3DRENDERSTATESINGLE**) As **Single**

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

state

Device state variable that is being queried. This parameter can be any of the constants of the **CONST_D3DRENDERSTATESINGLE** enumeration.

Return Values

If the method succeeds, the return value is the current value of the specified render state.

IDH_Direct3DDevice7.GetRenderStateSingle_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the parameters is invalid.

See Also

Direct3DDevice7.SetRenderState, **Direct3DDevice7.SetRenderStateSingle**

Direct3DDevice7.GetRenderTarget

The **Direct3DDevice7.GetRenderTarget** method retrieves the **DirectDrawSurface7** object that is being used as a render target.

object.**GetRenderTarget()** As **DirectDrawSurface7**

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

Return Values

If the method succeeds, a reference to the render-target surface (a **DirectDrawSurface7** object) is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set to DDERR_INVALIDPARAMS if one of the parameters is invalid.

See Also

Direct3DDevice7.SetRenderTarget

Direct3DDevice7.GetTexture

The **Direct3DDevice7.GetTexture** method retrieves a texture assigned to a given stage for a device.

object.**GetTexture(stage As Long)** As **DirectDrawSurface7**

IDH_Direct3DDevice7.GetRenderTarget_d3d_vb

IDH_Direct3DDevice7.GetTexture_d3d_vb

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

stage

Stage identifier of the texture to be retrieved. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value for *stage* is 7.

Return Values

If the method succeeds, the return value is the specified texture's **DirectDrawSurface7** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice7.SetTexture, **Direct3DDevice7.GetTextureStageState**, **Direct3DDevice7.SetTextureStageState**, Textures

Direct3DDevice7.GetTextureFormatsEnum

The **Direct3DDevice7.GetTextureFormatsEnum** method retrieves a **Direct3DEnumPixelFormatFormats** object.

object.**GetTextureFormatsEnum()** As **Direct3DEnumPixelFormatFormats**

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

IDH_Direct3DDevice7.GetTextureFormatsEnum_d3d_vb

Return Values

If the method succeeds, a reference to a **Direct3DEnumPixelFormat**s object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Use the methods of the **Direct3DEnumPixelFormat**s object to retrieve information about the pixel formats that the device supports for textures.

Direct3DDevice7.GetTextureStageState

The **Direct3DDevice7.GetTextureStageState** method retrieves a state value for a currently assigned texture.

```
object.GetTextureStageState( _  
    stage As Long, _  
    state As CONST_D3DTEXTURESTAGESTATETYPE) As Long
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

stage

Stage identifier of the texture for which the state will be retrieved. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value for *stage* is 7.

state

Texture state to be retrieved. This parameter can be any constant of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

Return Values

If the method succeeds, the return value is the state value.

IDH_Direct3DDevice7.GetTextureStageState_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice7.SetTextureStageState, **Direct3DDevice7.GetTexture**,
Direct3DDevice7.SetTexture, Textures

Direct3DDevice7.GetTextureStageStateSingle

The **Direct3DDevice7.GetTextureStageStateSingle** method retrieves a state value for a currently assigned texture, as type Single.

object.**GetTextureStageStateSingle**(_
 stage As Long, _
 state As CONST_D3DTEXTURESTAGESINGLE) As Single

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

stage

Stage identifier of the texture for which the state will be retrieved. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value for *stage* is 7.

state

Texture stage state to be retrieved. This parameter can be any constant of the **CONST_D3DTEXTURESTAGESINGLE** enumeration.

Return Values

If the method succeeds, the return value is the state value.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

IDH_Direct3DDevice7.GetTextureStageStateSingle_d3d_vb

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice7.SetTextureStageState,
Direct3DDevice7.SetTextureStageStateSingle, **Direct3DDevice7.GetTexture**,
Direct3DDevice7.SetTexture, Textures

Direct3DDevice7.GetTransform

The **Direct3DDevice7.GetTransform** method gets a matrix describing a transformation state.

```
object.GetTransform( _  
    transformType As CONST_D3DTRANSFORMSTATETYPE, _  
    matrix As D3DMATRIX)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

transformType

Device state variable that is being modified. This parameter can be any of the constants of the **CONST_D3DTRANSFORMSTATETYPE** enumeration.

matrix

A **D3DMATRIX** type describing the transformation.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the parameters is invalid.

See Also

Direct3DDevice7.SetTransform

Direct3DDevice7.GetViewport

The **Direct3DDevice7.GetViewport** method retrieves the viewport parameters currently set for the device.

IDH_Direct3DDevice7.GetTransform_d3d_vb

IDH_Direct3DDevice7.GetViewport_d3d_vb

object.**GetViewport**(_
 Viewport As D3DVIEWPORT7)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

Viewport

A **D3DVIEWPORT7** type that will be filled with the current viewport parameters if the call succeeds.

Return Values

If the method fails, an error is raised and **Err.Number** may be set to **DDERR_INVALIDPARAMS** if the *Viewport* parameter is invalid.

See Also

Direct3DDevice7.SetViewport

Direct3DDevice7.LightEnable

The **Direct3DDevice7.LightEnable** method enables or disables a set of lighting parameters within a device.

object.**LightEnable**(_
 LightIndex As Long, _
 bEnable As Boolean)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

LightIndex

Zero-based index of the set of lighting parameters that are the target of this method.

bEnable

Value indicating if the set of lighting parameters are being enabled or disabled. Set this parameter to nonzero to enable lighting with the parameters at the specified index or zero to disable it.

Error Codes

If the method fails, an error is raised and **Err.Number** is set to an error code. For a complete list of Direct3D Immediate Mode error values and descriptions, see Error Codes.

Remarks

If you supply a value for *LightIndex* outside the range of the light property sets currently assigned within the device, the **LightEnable** method creates a light source with the following properties and sets its enabled state to the value specified in *Enable*:

Member	Default
dltType	D3DLIGHT_DIRECTIONAL
dcvDiffuse	(R:1, G:1, B:1, A:0)
dcvSpecular	(R:0, G:0, B:0, A:0)
dcvAmbient	(R:0, G:0, B:0, A:0)
dvPosition	(0, 0, 0)
dvDirection	(0, 0, 1)
dvRange	0
dvFalloff	0
dvAttenuation0	0
dvAttenuation1	0
dvAttenuation2	0
dvTheta	0
dvPhi	0

See Also

Direct3DDevice7.GetLight, **Direct3DDevice7.GetLightEnable**, **Direct3DDevice7.SetLight**, **Lighting and Materials**

Direct3DDevice7.Load

The **Direct3DDevice7.Load** method loads a rectangular area of a source texture to a specified point in a destination texture or to faces within a cubic environment map.

```
object.Load( _
    DestTex As DirectDrawSurface7, _
    xDest As Long, _
    yDest As Long, _
    SrcTex As DirectDrawSurface7, _
```

IDH_Direct3DDevice7.Load_d3d_vb

rcSrcRect **As RECT**, *_*
flags **As Long**)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

DestTex

The **DirectDrawSurface7** class that will receive image data from the source texture. For complex texture surfaces like cubic environment maps and mipmaps, this must refer to the top-level surface.

xDest and *yDest*

The x- and y-coordinates of the point at which the method should load the image data onto the destination texture.

SrcTex

The **DirectDrawSurface7** class that contains the image data to be loaded. For complex texture surfaces like cubic environment maps and mipmaps, this must refer to the top-level surface.

rcSrcRect

Address of a **RECT** type that describes the area within the source texture that the method will load. Set this parameter to Nothing when the source rectangle should cover the entire source texture or when using this method to explicitly load a managed texture.

flags

Value indicating, in the case of a cubic environment map, which face of the destination texture is to receive the image data. This can be any combination of the following flags.

0

Required if the destination texture is a managed texture.

DDSCAPS2_CUBEMAP_ALLFACES

All faces should be loaded with the image data within the source texture.

DDSCAPS2_CUBEMAP_NEGATIVEX,

DDSCAPS2_CUBEMAP_NEGATIVEY, or

DDSCAPS2_CUBEMAP_NEGATIVEZ

The negative x, y, or z faces should receive the image data.

DDSCAPS2_CUBEMAP_POSITIVEX,

DDSCAPS2_CUBEMAP_POSITIVEY, or

DDSCAPS2_CUBEMAP_POSITIVEZ

The positive x, y, or z faces should receive the image data.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

The destination texture can be used with any rendering device, not just the one that created it. On hardware devices, the load operation is hardware-accelerated.

Loading textures into video memory by calling this method is preferred over blit operations.

This method copies mipmaps at all levels, cube-map faces, palettes and color keys from the source texture to the destination texture.

When using this method to load mipmaps, the following special points apply:

- The destination texture surface can be a subset of the mip-levels contained in the source texture. In this case, only the mip-levels common to both textures are copied. This is useful in implementing texture LOD. You can create a video memory texture with only two mip-levels, then use this method to copy those levels from a source texture that comprises the entire set of mip-levels.
- The source and destination surface pointers must point to top-level surfaces.
- When loading a subrectangle of a mipmap, the subrectangle refers to the top-level surface in the mipmap chain, and is divided by two for each lower mip-level.

Direct3DDevice7.MultiplyTransform

The **Direct3DDevice7.MultiplyTransform** method multiplies a device's world, view, or projection matrices by a specified matrix. The multiplication order is *matrix* times *dstTransformStateType*.

object.**MultiplyTransform**(
 dstTransformStateType As CONST_D3DTRANSFORMSTATETYPE,
 matrix As D3DMATRIX)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

dstTransformStateType

Identifies which device matrix is to be modified. This parameter can be any of the constants of the **CONST_D3DTRANSFORMSTATETYPE** enumeration. The most common setting, **D3DTRANSFORMSTATE_WORLD**, modifies the

IDH_Direct3DDevice7.MultiplyTransform_d3d_vb

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 DDERR_INVALIDPARAMS if one of the parameters is invalid.

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.)

```
Direct3DDevice7.SetTransform(D3DTRANSFORMSTATE_WORLD,
    shoulder_transform)
Direct3DDevice7.DrawPrimitive(upper_arm)
Direct3DDevice7.MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
    elbow_transform)
Direct3DDevice7.DrawPrimitive(lower_arm)
Direct3DDevice7.MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
    wrist_transform)
Direct3DDevice7.DrawPrimitive(hand)
```

See Also

Direct3DDevice7.DrawPrimitive, **Direct3DDevice7.SetTransform**

Direct3DDevice7.PreLoad

The **Direct3DDevice7.Preload** method instructs the Direct3D texture manager to load a managed texture into video memory.

IDH_Direct3DDevice7.PreLoad_d3d_vb

object.PreLoad(_
Texture As DirectDrawSurface7)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

Texture

Reference to the **DirectDrawSurface7** object to be loaded into memory.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method forces the system to load a managed texture into video memory. If sufficient video memory is not available, the system evicts other textures to regain memory, then loads the texture. The texture being loaded must be a managed texture (created with the DDSCAPS2_TEXTUREMANAGE or DDSCAPS2_D3DTEXTUREMANAGE flags). If these requirements are not met, the method fails and returns DDERR_INVALIDPARAMS.

See Also

Automatic Texture Management

Direct3DDevice7.SetClipPlane

The **Direct3DDevice7.SetClipPlane** method sets the coefficients of a user-defined clipping plane for the device.

object.SetClipPlane(_
Index As Long, _
a As Single, _
b As Single, _
c As Single, _
d As Single)

IDH_Direct3DDevice7.SetClipPlane_d3d_vb

Parameters

Index

Index of the clipping plane for which the plane equation coefficients are retrieved.

a, b, c, and d

Values that represent the coefficients of the clipping plane, 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 DDERR_INVALIDPARAMS. This error indicates that the value in *Index* exceeds the maximum clipping plane index supported by the device.

Remarks

The coefficients that this method reports take the form of the general plane equation. The values in *a*, *b*, *c*, and *d* would 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 $Ax + By + Cz + Dw \geq 0$. 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.

This method does not enable the clipping plane equation being set. To enable a clipping plane, use the D3DRENDERSTATE_CLIPPLANEENABLE render state.

See Also

Direct3DDevice7.GetClipPlane, D3DRENDERSTATE_CLIPPLANEENABLE, User-defined Clip Planes

Direct3DDevice7.SetClipStatus

The **Direct3DDevice7.SetClipStatus** method sets the current clip status.

object.SetClipStatus(*clipStatus* As D3DCLIPSTATUS)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

clipStatus

A **D3DCLIPSTATUS** type that describes the new settings for the clip status.

IDH_Direct3DDevice7.SetClipStatus_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if the *clipStatus* parameter is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice7.GetClipStatus

Direct3DDevice7.SetLight

The **Direct3DDevice7.SetLight** method assigns a set of lighting properties for this device.

```
object.SetLight( _  
    LightIndex As Long, _  
    Light As D3DLIGHT7)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

LightIndex

Zero-based index of the set of lighting properties to be set. If a set of lighting properties already exists at this index, it will be overwritten by the new properties.

Light

A **D3DLIGHT7** type that contains the lighting parameters to be set.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

```
DDERR_INVALIDPARAMS  
DDERR_OUTOFMEMORY
```

See Also

Direct3DDevice7.GetLight, **Direct3DDevice7.GetLightEnable**,
Direct3DDevice7.LightEnable , Lighting and Materials.

IDH_Direct3DDevice7.SetLight_d3d_vb

Direct3DDevice7.SetMaterial

The **Direct3DDevice7.SetMaterial** method sets the material properties for the device.

object.SetMaterial(_
 mat As D3DMATERIAL7)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

mat

A **D3DMATERIAL7** type that describes the material properties to be set.

Error Codes

If the method succeeds, the return value is **D3D_OK**.

If the method fails, the return value is an error. The method returns **DDERR_INVALIDPARAMS** if the *mat* parameter is invalid.

See Also

Direct3DDevice7.GetMaterial

Direct3DDevice7.SetRenderState

The **Direct3DDevice7.SetRenderState** method sets a value (of type **Long**) in a device render state.

object.SetRenderState(_
 state As CONST_D3DRENDERSTATETYPE, _
 renderstate As Long)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

state

Device state variable that is being modified. This parameter can be any of the constants of the **CONST_D3DRENDERSTATETYPE** enumeration.

renderstate

IDH_Direct3DDevice7.SetMaterial_d3d_vb

IDH_Direct3DDevice7.SetRenderState_d3d_vb

New render-state value. The meaning of this parameter is dependent on the value specified for *state*. For example, if *state* were `D3DRENDERSTATE_SHADEMODE`, the second parameter would be one of the constants 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 `DDERR_INVALIDPARAMS` if one of the parameters is invalid.

See Also

`Direct3DDevice7.GetRenderState`, `Direct3DDevice7.SetTransform`

Direct3DDevice7.SetRenderStateSingle

The `Direct3DDevice7.SetRenderStateSingle` method sets a value (of type **Single**) in a device render state.

```
object.SetRenderStateSingle( _
    state As CONST_D3DRENDERSTATESINGLE, _
    renderstate As Single)
```

Parameters

object

Object expression that resolves to a `Direct3DDevice7` object.

state

Device state variable that is being modified. This parameter can be any of the constants of the `CONST_D3DRENDERSTATESINGLE` enumeration.

renderstate

New render-state value. The meaning of this parameter is dependent on the value specified for *state*. For example, if *state* were `D3DRENDERSTATE_SHADEMODE`, the second parameter would be one of the constants 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 `DDERR_INVALIDPARAMS` if one of the parameters is invalid.

See Also

Direct3DDevice7.GetRenderState, **Direct3DDevice7.GetRenderStateSingle**,
Direct3DDevice7.SetTransform

Direct3DDevice7.SetRenderTarget

The **Direct3DDevice7.SetRenderTarget** method permits the application to easily route rendering output to a new DirectDraw surface as a render target.

object.SetRenderTarget(surface As DirectDrawSurface7)

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

surface

A **DirectDrawSurface7** object for the previously created surface object that will be the new rendering target.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set one of the following values:

DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Capabilities do not change with changes in the properties of the render target surface. Both the Direct3D HAL and the software rasterizers have only one opportunity to expose capabilities to the application. The system cannot expose different sets of capabilities depending on the format of the destination surface.

If a depth-buffer is attached to the new render target, it replaces the previous z-buffer for the context. Otherwise, the old z-buffer is detached and z-buffering is disabled.

If more than one depth-buffer is attached to the render target, this function fails.

See Also

Direct3DDevice7.GetRenderTarget

IDH_Direct3DDevice7.SetRenderTarget_d3d_vb

Direct3DDevice7.SetTexture

The **Direct3DDevice7.SetTexture** method assigns a texture to a given stage for a device.

```
object.SetTexture( _  
    stage As Long, _  
    texture As DirectDrawSurface7)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

stage

Stage identifier to which the texture will be set. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value for *stage* is 7.

texture

A **DirectDrawSurface7** object for the texture being set. For complex textures, such as mipmaps and cubic environment maps, this parameter must be the top-level surface.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

```
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS
```

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

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.

Software devices do not support assigning a texture to more than one texture stage at a time.

See Also

Direct3DDevice7.GetTexture, **Direct3DDevice7.GetTextureStageState**, **Direct3DDevice7.SetTextureStageState**, Textures

IDH_Direct3DDevice7.SetTexture_d3d_vb

Direct3DDevice7.SetTextureStageState

The **Direct3DDevice7.SetTextureStageState** method sets the state value for a currently assigned texture.

```
object.SetTextureStageState( _  
    stage As Long, _  
    state As CONST_D3DTEXTURESTAGESTATETYPE, _  
    value As Long)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

stage

Stage identifier of the texture for which the state value will be set. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value for *stage* is 7.

state

Texture state to be set. This parameter can be any constant of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

value

State value to be set. The meaning of this value is determined by the *state* parameter.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

```
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS
```

Remarks

Applications should use this method to set texture states in favor of the legacy texture-related render states. For more information, see [About Render States](#).

See Also

Direct3DDevice7.GetTextureStageState, **Direct3DDevice7.GetTexture**, **Direct3DDevice7.SetTexture**, [Textures](#)

IDH_Direct3DDevice7.SetTextureStageState_d3d_vb

Direct3DDevice7.SetTextureStageStateSingle

The **Direct3DDevice7.SetTextureStageStateSingle** method sets the state value for a currently assigned texture.

```
object.SetTextureStageStateSingle( _  
    stage As Long, _  
    state As CONST_D3DTEXTURESTAGESINGLE, _  
    value As Single)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

stage

Stage identifier of the texture for which the state value will be set. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value for *stage* is 7.

state

Texture state to be set. This parameter can be any constant of the **CONST_D3DTEXTURESTAGESINGLE** enumeration.

value

State value to be set. The meaning of this value is determined by the *state* parameter.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

```
DDERR_INVALIDOBJECT  
DDERR_INVALIDPARAMS
```

Remarks

Applications should use this method to set texture states in favor of the legacy texture-related render states. For more information, see About Render States.

See Also

Direct3DDevice7.GetTextureStageStateSingle,
Direct3DDevice7.GetTextureStageState, **Direct3DDevice7.GetTexture**,
Direct3DDevice7.SetTexture, Textures

IDH_Direct3DDevice7.SetTextureStageStateSingle_d3d_vb

Direct3DDevice7.SetTransform

The **Direct3DDevice7.SetTransform** method sets a single transformation-related state for the device.

```
object.SetTransform( _  
    transformType As CONST_D3DTRANSFORMSTATETYPE, _  
    matrix As D3DMATRIX)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

transformType

Device state variable that is being modified. This parameter can be any of the constants of the **CONST_D3DTRANSFORMSTATETYPE** enumeration.

matrix

A **D3DMATRIX** type that modifies the current transformation.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to **DDERR_INVALIDPARAMS** if one of the parameters is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice7.GetTransform, **Direct3DDevice7.SetRenderState**

Direct3DDevice7.SetViewport

The **Direct3DDevice7.SetViewport** method sets the viewport parameters for the device.

```
object.SetViewport( _  
    viewport As D3DVIEWPORT7)
```

Parameters

object

Object expression that resolves to a **Direct3DDevice7** object.

viewport

A **D3DVIEWPORT7** type that contains the viewport parameters to be set.

IDH_Direct3DDevice7.SetTransform_d3d_vb

IDH_Direct3DDevice7.SetViewport_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS to indicate that the *viewport* parameter is invalid.

Remarks

If the *viewport* parameters described by the **D3DVIEWPORT7** type describe a region that cannot exist within the render target surface, the method will fail, returning DDERR_INVALIDPARAMS.

See Also

Direct3DDevice7.GetViewport

Direct3DDevice7.ValidateDevice

The **Direct3DDevice7.ValidateDevice** method reports the device's ability to render the currently set texture blending operations and parameters in a single pass.

object.**ValidateDevice()** As Long

Parameters

object

Object expression that resolves to a **Direct3DDevice7** 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** may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
D3DERR_CONFLICTINGTEXTUREFILTER
D3DERR_TOOMANYOPERATIONS
D3DERR_UNSUPPORTEDALPHAARG
D3DERR_UNSUPPORTEDALPHAOPERATION

IDH_Direct3DDevice7.ValidateDevice_d3d_vb

D3DERR_UNSUPPORTEDCOLORARG
D3DERR_UNSUPPORTEDCOLOROPERATION
D3DERR_UNSUPPORTEDFACTORVALUE
D3DERR_UNSUPPORTEDTEXTUREFILTER
D3DERR_WRONGTEXTUREFORMAT

Remarks

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 parameters by setting-up the desired blending operation, then calling the **ValidateDevice** method.

The **ValidateDevice** method uses the currently set render states, textures, and, texture stage states to perform validation at the time of the call. Any 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 sparsely supported on current hardware. Most hardware can only introduce iterated color data at the last texture operation stage.

Try to specify the texture (D3DTA_TEXTURE) for each stage as the first parameter, in preference to the second parameter.

Many cards do not support use of diffuse or scalar values at arbitrary texture stages. Often, these are only available at the first or last texture blending stage.

Many cards do not actually have a blending unit associated with the first texture that is capable of more than replicating alpha to color channels, or inverting the input. As a result, 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 parameter 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 both 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, it might be best to perform multipass rendering instead.

See Also

Direct3DDevice7.GetTextureStageState, **Direct3DDevice7.SetTextureStageState**

Direct3DEnumDevices

Applications use the methods of the **Direct3DEnumDevices** class to retrieve information about the Direct3D devices present on a system. The **Direct3DEnumDevices** class is obtained by calling the **Direct3D7.GetDevicesEnum** method.

The methods of the **Direct3DEnumDevices** class can be organized into the following groups:

Device count	GetCount
Device information	GetDescription
	GetGuid
	GetDesc
	GetName

Direct3DEnumDevices.GetCount

The **Direct3DEnumDevices.GetCount** method returns the number of Direct3D devices installed on the system.

object.GetCount() As Long

Parameters

object

Object expression that resolves to a **Direct3DEnumDevices** object.

Return Values

If the method succeeds, the number of Direct3D devices installed on the system is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DEnumDevices.GetDesc

The **Direct3DEnumDevices.GetDesc** method returns the capabilities of the desired Direct3D device.

object.GetDesc(*index* As Long, *hwDesc* As D3DDEVICEDESC7)

IDH_Direct3DEnumDevices_d3d_vb

IDH_Direct3DEnumDevices.GetCount_d3d_vb

IDH_Direct3DEnumDevices.GetDesc_d3d_vb

Parameters

object

Object expression that resolves to a **Direct3DEnumDevices** object.

index

On a system with multiple Direct3D devices, this parameter represents the specific device.

hwDesc

A **D3DDEVICEDESC7** type that contains the hardware capabilities of the Direct3D device.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DEnumDevices.GetDescription

The **Direct3DEnumDevices.GetDescription** method returns user-friendly description of the desired Direct3D device.

object.**GetDescription**(*index* As Long) As String

Parameters

object

Object expression that resolves to a **Direct3DEnumDevices** object.

index

Index of the enumerated device. This value can be between 1 and the value returned by the **Direct3DEnumDevices.GetCount** method.

Return Values

If the method succeeds, the return value is the user-friendly description of the enumerated device.

Error Codes

If the method fails, the method sets **Err.Number** to an error code and raises an error. For a list of possible error codes, see Direct3D Immediate Mode Error Codes.

Direct3DEnumDevices.GetGuid

The **Direct3DEnumDevices.GetDescription** method returns the globally-unique ID (GUID) of the desired Direct3D device.

object.**GetGuid**(*index* As Long) As String

Parameters

object

Object expression that resolves to a **Direct3DEnumDevices** object.

index

Index of the enumerated device. This value can be between 1 and the value returned by the **Direct3DEnumDevices.GetCount** method.

Return Values

If the method succeeds, the return value is a string label—"IID_IDirect3DHALDevice", "IID_IDirect3DTnLHalDevice", or "IID_IDirect3DRGBDevice"—that represents the globally-unique identifier of the enumerated device. This value is used to create the device with a subsequent call to the **Direct3D7.CreateDevice** method.

Error Codes

If the method fails, the method sets **Err.Number** to an error code and raises an error. For a list of possible error codes, see Direct3D Immediate Mode Error Codes.

Direct3DEnumDevices.GetName

The **Direct3DEnumDevices.GetName** method returns the user-friendly name of the desired Direct3D device.

object.**GetName**(*index* As Long) As String

Parameters

object

Object expression that resolves to a **Direct3DEnumDevices** object.

index

Index of the enumerated device. This value can be between 1 and the value returned by the **Direct3DEnumDevices.GetCount** method.

IDH_Direct3DEnumDevices.GetGuid_d3d_vb

IDH_Direct3DEnumDevices.GetName_d3d_vb

Return Values

If the method succeeds, the return value is the user-friendly name of the device.

Error Codes

If the method fails, the method sets **Err.Number** to an error code and raises an error. For a list of possible error codes, see Direct3D Immediate Mode Error Codes.

Direct3DEnumPixelFormatFormats

Applications use the methods of the **Direct3DEnumPixelFormatFormats** class to retrieve information about the pixel formats supported by a rendering device. The **Direct3DEnumPixelFormatFormats** class is obtained by calling the **Direct3DDevice7.GetTextureFormatsEnum** and **Direct3D7.GetEnumZBufferFormats** methods.

This class consists of two methods.

- **GetCount**
- **GetItem**

Direct3DEnumPixelFormatFormats.GetCount

The **Direct3DEnumPixelFormatFormats.GetCount** method returns the number of supported pixel formats of the Direct3D device.

object.**GetCount()** As Long

Parameters

object

Object expression that resolves to a **Direct3DEnumPixelFormatFormats** object.

Return Values

If the method succeeds, the number of supported pixel formats is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

IDH_Direct3DEnumPixelFormatFormats_d3d_vb
IDH_Direct3DEnumPixelFormatFormats.GetCount_d3d_vb

Direct3DEnumPixelFormat.GetItem

The **Direct3DEnumPixelFormat.GetItem** method returns the description of the specified pixel format.

object.**GetItem**(*index* As Long, _
pixelFormat As DDPIXELFORMAT)

Parameters

object

Object expression that resolves to a **Direct3DEnumPixelFormat** object.

index

The specific pixel format that you want a description for.

pixelFormat

A **DDPIXELFORMAT** type that describes the enumerated pixel format.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DVertexBuffer7

Applications use the methods of the **Direct3DVertexBuffer7** class to manipulate a collection of vertices for use with the **Direct3DDevice7.DrawPrimitiveVB** and **Direct3DDevice7.DrawIndexedPrimitiveVB** rendering methods. This section is a reference to the methods of this class. For a conceptual overview, see Vertex Buffers.

This methods of the **Direct3DVertexBuffer7** class can be organized into the following groups:

Information

Vertex data

GetVertexBufferDesc

GetVertices

Lock

Optimize

ProcessVertices

SetVertices

Unlock

See Also

Vertex Buffers

IDH_Direct3DEnumPixelFormat.GetItem_d3d_vb

IDH_Direct3DVertexBuffer7_d3d_vb

Direct3DVertexBuffer7.GetVertexBufferDesc

The **Direct3DVertexBuffer7.GetVertexBufferDesc** method retrieves a description of the vertex buffer.

```
object.GetVertexBufferDesc(desc As D3DVERTEXBUFFERDESC)
```

Parameters

object

Object expression that resolves to a **Direct3DVertexBuffer7** object.

desc

A **D3DVERTEXBUFFERDESC** type that will be filled with a description of the vertex buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be DDERR_INVALIDPARAMS or another error code.

Direct3DVertexBuffer7.GetVertices

The **Direct3DVertexBuffer7.GetVertices** method returns the vertices in the vertex buffer.

```
object.GetVertices(startIndex As Long, _  
    count As Long, _  
    verts As Any)
```

Parameters

object

Object expression that resolves to a **Direct3DVertexBuffer7** object.

startIndex

Starting index within the vertex buffer at which vertices will be retrieved.

count

Number of vertices to be retrieved.

verts

First element of an array that will be filled with the retrieved vertices.

IDH_Direct3DVertexBuffer7.GetVertexBufferDesc_d3d_vb

IDH_Direct3DVertexBuffer7.GetVertices_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DVertexBuffer7.Lock

The **Direct3DVertexBuffer7.Lock** methods locks a vertex buffer.

object.Lock(flags As CONST_DDLOCKFLAGS)

Parameters

object

Object expression that resolves to a **Direct3DVertexBuffer7** object.

flags

One of the following constants from the **CONST_DDLOCKFLAGS** enumeration indicating how the vertex buffer memory should be locked.

DDLOCK_NOSYSLOCK

If possible, do not take the Win16Mutex (also known as Win16Lock).

DDLOCK_READONLY

Indicates that the memory being locked will only be read from.

DDLOCK_WAIT

If a lock cannot be obtained immediately, the method retries until a lock is obtained or another error occurs.

DDLOCK_WRITEONLY

Indicates that the memory being locked will only be written to.

DDLOCK_DISCARDCONTENTS

Used only with Direct3D vertex-buffer locks. Indicates that no assumptions are made about the contents of the vertex buffer during this lock. This enables Direct3D or the driver to provide an alternative memory area as the vertex buffer. This is useful when you plan to clear the contents of the vertex buffer and fill in new data.

DDLOCK_NOOVERWRITE

Used only with Direct3D vertex-buffer locks. Indicates that no vertices that were referred to in vertex-buffer DrawPrimitive calls since the start of the frame (or the last lock without this flag) are modified during the lock. This can be useful when you want only to append data to the vertex buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_VERTEXBUFFEROPTIMIZED

IDH_Direct3DVertexBuffer7.Lock_d3d_vb

DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACEBUSY
DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

While the vertex buffer is locked, you can manipulate the data within it by calling the **Direct3DVertexBuffer7.GetVertices** and **Direct3DVertexBuffer7.SetVertices** methods. After locking the vertex buffer, you can access the memory until a corresponding call to **Direct3DVertexBuffer7.Unlock**.

You cannot render from a locked vertex buffer; calls to the **Direct3DDevice7.DrawIndexedPrimitiveVB** or **Direct3DDevice7.DrawPrimitiveVB** method using a locked buffer will fail, returning D3DERR_VERTEXBUFFERLOCKED.

This method often causes the system to hold the Win16Mutex until you call the **Direct3DVertexBuffer7.Unlock** method. GUI debuggers cannot operate while the Win16Mutex is held.

See Also

Direct3DVertexBuffer7.Unlock, Accessing the Contents of a Vertex Buffer

Direct3DVertexBuffer7.Optimize

The **Direct3DVertexBuffer7.Optimize** method converts an unoptimized vertex buffer into an optimized vertex buffer.

object.Optimize(dev As Direct3DDevice7)

Parameters

object

Object expression that resolves to a **Direct3DVertexBuffer7** object.

dev

A **Direct3DDevice7** object of the device for which this vertex buffer will be optimized.

IDH_Direct3DVertexBuffer7.Optimize_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_VERTEXBUFFEROPTIMIZED
 D3DERR_VERTEXBUFFERLOCKED
 DDERR_INVALIDPARAMS
 DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Optimizing a Vertex Buffer, Vertex Buffers

Direct3DVertexBuffer7.ProcessVertices

The **Direct3DVertexBuffer7.ProcessVertices** method processes untransformed vertices into a transformed or optimized vertex buffer.

```
object.ProcessVertices( _  
    vertexOp As CONST_D3DVOPFLAGS, _  
    destIndex As Long, _  
    count As Long, _  
    srcBuffer As Direct3DVertexBuffer7, _  
    srcIndex As Long, _  
    dev As Direct3DDevice7, _  
    flags As CONST_D3DPROCESSVERTICESFLAGS)
```

Parameters

object

Object expression that resolves to a **Direct3DVertexBuffer7** object.

vertexOp

Flags defining how the method processes the vertices as they are transferred from the source buffer. You can specify any combination of the constants from the **CONST_D3DVOPFLAGS** enumeration:

destIndex

Index into the destination vertex buffer (this buffer) where the vertices will be placed after processing.

count

Number of vertices in the source buffer to process.

srcBuffer

A **Direct3DVertexBuffer7** object for the source vertex buffer.

IDH_Direct3DVertexBuffer7.ProcessVertices_d3d_vb

srcIndex

Index of the first vertex in the source buffer to be processed.

dev

A **Direct3DDevice7** object for the device that will be used to transform the vertices.

flags

Processing options. Set this parameter to one of the members of the **CONST_D3DPROCESSVERTICESFLAGS** enumeration.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDVERTEXFORMAT
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACEBUSY
DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

You must always include the D3DVOP_TRANSFORMED flag in the *vertexOp* parameter. If you fail to include this flag, the method will fail, returning DDERR_INVALIDPARAMS.

See Also

Processing Vertices, Vertex Buffers

Direct3DVertexBuffer7.SetVertices

The **Direct3DVertexBuffer7.SetVertices** method sets the vertices in a locked vertex buffer.

object.**SetVertices**(*startIndex* As Long, _
 count As Long, _
 verts As Any)

Parameters

object

IDH_Direct3DVertexBuffer7.SetVertices_d3d_vb

Object expression that resolves to a **Direct3DVertexBuffer7** object.

startIndex

Zero-based index within the vertex buffer at which the vertices will be set.

count

Number of vertices contained in the *verts* array.

verts

First element of the buffer containing the vertex data.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set to an error code. For a complete list of Direct3D Immediate Mode error values and descriptions, see Error Codes.

Remarks

You must lock a vertex buffer by calling the **Direct3DVertexBuffer7.Lock** method before manipulating the vertices it contains.

See Also

Direct3DVertexBuffer7.GetVertices, **Direct3DVertexBuffer7.Lock**

Direct3DVertexBuffer7.Unlock

The **Direct3DVertexBuffer7.Unlock** method unlocks a previously locked vertex buffer.

object.**Unlock()**

Parameters

object

Object expression that resolves to a **Direct3DVertexBuffer7** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACEBUSY

IDH_Direct3DVertexBuffer7.Unlock_d3d_vb

DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DVertexBuffer7.Lock, Accessing the Contents of a Vertex Buffer

Types

This section contains information about the following types used with Direct3D Immediate Mode.

- **D3DCLIPSTATUS**
- **D3DCOLORVALUE**
- **D3DDEVICEDESC7**
- **D3DDEVINFO_TEXTUREMANAGER**
- **D3DDEVINFO_TEXTUREING**
- **D3DLIGHT7**
- **D3DLIGHTINGCAPS**
- **D3DLINEPATTERN**
- **D3DLVERTEX**
- **D3DMATERIAL7**
- **D3DMATRIX**
- **D3DPRIMCAPS**
- **D3DRECT**
- **D3DTLVERTEX**
- **D3DVECTOR**
- **D3DVERTEX**
- **D3DVERTEXBUFFERDESC**
- **D3DVIEWPORT7**
- **DXDRIVERINFO**

D3DCLIPSTATUS

The **D3DCLIPSTATUS** type describes the current clip status and extents of the clipping region.

Type D3DCLIPSTATUS
IFlags As Long
IStatus As Long

IDH_D3DCLIPSTATUS_d3d_vb

maxx As Single
 maxy As Single
 maxz As Single
 minx As Single
 miny As Single
 minz As Single
 End Type

Members

IFlags

Flags describing whether this type describes 2-D extents, 3-D extents, or the clip status. This member can be a combination of the following flags from the **CONST_D3DCLIPSTATUSFLAGS** enumeration:

D3DCLIPSTATUS_STATUS

The type describes the current clip status.

D3DCLIPSTATUS_EXTENTS2

The type describes the current 2-D extents. This flag cannot be combined with **D3DCLIPSTATUS_EXTENTS3**.

D3DCLIPSTATUS_EXTENTS3

Not currently implemented.

IStatus

Describes the current clip status. This member can be one or more of the following constants of the **CONST_D3DCLIPFLAGS** enumeration:

Combination and General Flags

D3DSTATUS_CLIPINTERSECTIONALL

Combination of all **CLIPINTERSECTION** flags.

D3DSTATUS_CLIPUNIONALL

Combination of all **CLIPUNION** flags.

D3DSTATUS_DEFAULT

Combination of **D3DSTATUS_CLIPINTERSECTIONALL** and **D3DSTATUS_ZNOTVISIBLE** flags. This value is the default.

D3DSTATUS_ZNOTVISIBLE

Indicates that the rendered primitive is not visible. This flag is set or cleared by the system when rendering with z-checking enabled (see **D3DRENDERSTATE_ZVISIBLE**).

Clip Intersection Flags

D3DSTATUS_CLIPINTERSECTIONBACK

Logical **And** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM

Logical **And** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT

Logical **And** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through**D3DSTATUS_CLIPINTERSECTIONGEN5**

Logical **And** of the clip flags for application-defined clipping planes.

D3DSTATUS_CLIPINTERSECTIONLEFT

Logical **And** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT

Logical **And** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP

Logical **And** of the clip flags for the vertices compared to the top of the viewing frustum.

Clip Union Flags**D3DSTATUS_CLIPUNIONBACK**

Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM

Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT

Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5

Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT

Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT

Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP

Equal to D3DCLIP_TOP.

Basic Clipping Flags**D3DCLIP_BACK**

All vertices are clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM

All vertices are clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT

All vertices are clipped by the front plane of the viewing frustum.

D3DCLIP_LEFT

All vertices are clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT

All vertices are clipped by the right plane of the viewing frustum.

D3DCLIP_TOP

All vertices are clipped by the top plane of the viewing frustum.

D3DCLIP_GEN0 through D3DCLIP_GEN5

Application-defined clipping planes.

maxx, minx, maxy, miny, maxz, minz

The x, y, and z extents of the current clipping region.

See Also

Direct3DDevice7.GetClipStatus, **Direct3DDevice7.SetClipStatus**

D3DCOLORVALUE

The **D3DCOLORVALUE** type describes color values for the **D3DLIGHT7** and **D3DMATERIAL7** types.

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 range from 0 to 1, with 0 being black.

Remarks

You can set the members of this type to values outside the range of 0 to 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. For more information, see [Colored Lights](#).

D3DDEVICEDESC7

The **D3DDEVICEDESC7** type contains a description of the current device. This type is used to query the current device by such methods as **Direct3DDevice7.GetCaps**.

Type D3DDeviceDesc7

dpcLineCaps As D3DPRIMCAPS

dpcTriCaps As D3DPRIMCAPS

dvExtentsAdjust As Single

IDH_D3DCOLORVALUE_d3d_vb

IDH_D3DDEVICEDESC7_d3d_vb

dvGuardBandBottom As Single
 dvGuardBandLeft As Single
 dvGuardBandRight As Single
 dvGuardBandTop As Single
 dvMaxVertexW As Single
 IDevCaps As CONST_D3DDEVICEDESCCAPS
 IDeviceRenderBitDepth As Long
 IDeviceZBufferBitDepth As Long
 IFVFCaps As CONST_D3DFVFCAPSFLAGS
 IMaxActiveLights As Long
 IMaxAnisotropy As Long
 IMaxTextureAspectRatio As Long
 IMaxTextureHeight As Long
 IMaxTextureRepeat As Long
 IMaxTextureWidth As Long
 IMinTextureHeight As Long
 IMinTextureWidth As Long
 IStencilCaps As CONST_D3DSTENCILCAPSFLAGS
 ITextureOpCaps As CONST_D3DTEXOPCAPSFLAGS
 IVertexProcessingCaps As CONST_D3DVTXPCAPS
 nMaxSimultaneousTextures As Integer
 nMaxTextureBlendStages As Integer
 nMaxUserClipPlanes As Integer
 nMaxVertexBlendMatrices As Integer
 End Type

Members

dpcLineCaps and dpcTriCaps

D3DPRIMCAPS type defining the device's support for line-drawing and triangle primitives.

dvExtentsAdjust

Number of pixels to adjust the extents rectangle outward to accommodate antialiasing kernels.

dvGuardBandBottom, dvGuardBandLeft, dvGuardBandTop, and dvGuardBandRight

The screen-space coordinates of the guard-band clipping region. Coordinates inside this rectangle but outside the viewport rectangle will automatically be clipped.

dvMaxVertexW

Maximum W-based depth value that the device supports.

IDevCaps

One of the constants of the **CONST_D3DDEVICEDESCCAPS** enumeration identifying the capabilities of the device.

IDeviceRenderBitDepth

Device's rendering bit-depth. This can be one or more of the following constants from the **CONST_DDBITDEPTHFLAGS** enumeration: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

IDeviceZBufferBitDepth

Device's depth-buffer bit-depth. This can be one or more of the following constants from the **CONST_DDBITDEPTHFLAGS** enumeration: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

IFVFCaps

Combination of constants of the **CONST_D3DFVFCAPSFLAGS** that describe the vertex formats supported by this device.

IMaxActiveLights

Maximum number of lights that can be active simultaneously.

IMaxAnisotropy

Maximum valid value for D3DTSS_MAXANISOTROPY texture-stage state.

IMaxTextureAspectRatio

Maximum texture aspect ratio supported by the hardware; this will typically be a power of 2.

IMaxTextureWidth, IMaxTextureHeight

Maximum texture width and height for this device.

IMaxTextureRepeat

Full range of the integer (non-fractional) bits of the post-normalized texture indices. If the D3DPTEXTURECAPS_TEXREPEATNOTSCALEDDBYSIZE flag is set, the device defers scaling by the texture size until after the texture address mode is applied. If it isn't set, the device scales the texture indices by the texture size (largest level-of-detail) prior to interpolation.

IMinTextureWidth, IMinTextureHeight

Minimum texture width and height for this device.

IStencilCaps

Constants of the **CONST_D3DSTENCILCAPSFLAGS** enumeration specifying supported stencil-buffer operations. Stencil operations are assumed to be valid for all three stencil-buffer operation render states (D3DRENDERSTATE_STENCILFAIL, D3DRENDERSTATE_STENCILPASS, and D3DRENDERSTATE_STENCILFAILZFFAIL).

ITextureOpCaps

Combination of constants of the **CONST_D3DTEXOPCAPSFLAGS** enumeration describing the texture operations supported by this device.

IVertexProcessingCaps

Vertex processing capabilities, described as a combination of constants of the **CONST_D3DVTXPCAPS** enumeration.

nMaxSimultaneousTextures

Maximum number of textures that can be simultaneously bound to the texture blending stages for this device. See remarks.

nMaxTextureBlendStages

Maximum number of texture blending stages supported by this device.

nMaxUserClipPlanes

Maximum number of user-defined clipping planes supported. This member can range from 0 through 31.

User-defined clipping planes are manipulated by using the

Direct3DDevice7.GetClipPlane and **Direct3DDevice7.SetClipPlane** methods.

nMaxVertexBlendMatrices

Maximum number of matrices that this device can apply when performing geometry blending.

Remarks

The **nMaxTextureBlendStages** and **nMaxSimultaneousTextures** members might seem very similar at first glance, but they contain different information. The **nMaxTextureBlendStages** member contains the total number of texture-blending stages supported by the current device, and the **nMaxSimultaneousTextures** member describes how many of those stages can have textures bound to them by using the **Direct3DDevice7.SetTexture** method.

See Also

CONST_D3DCOLORMODEL, **D3DLIGHTINGCAPS**, **D3DPRIMCAPS**

D3DDEVINFO_TEXTUREMANAGER

The **D3DDEVINFO_TEXTUREMANAGER** type contains information about the current state of the texture manager. This structure is used with the **Direct3DDevice7.GetInfo** method.

Type **D3DDEVINFO_TEXTUREMANAGER**

bThrashing As Boolean
 lLastPri As Long
 lNumEvicts As Long
 lNumTexturesUsed As Long
 lNumUsedTexInVid As Long
 lNumVidCreates As Long
 lTotalBytes As Long
 lTotalManaged As Long
 lWorkingSet As Long
 lWorkingSetBytes As Long

End Type

IDH_D3DDEVINFO_TEXTUREMANAGER_d3d_vb

Members

bThrashing

Thrashing status. This member is True if thrashing occurred during the last frame, or False otherwise.

ILastPri

Priority of last evicted texture.

INumEvicts

Number of textures that were evicted during the last frame.

INumTexturesUsed

Total number of textures used during the last frame.

INumUsedTexInVid

Number of the video memory textures that were used during the last frame.

INumVidCreates

Number of textures that were created in video memory during the last frame.

ITotalBytes

Total number of bytes allocated for managed textures.

ITotalManaged

Total number of managed textures.

IWorkingSet

Number of textures currently resident in video memory.

IWorkingSetBytes

Number of bytes currently allocated by textures resident in video memory.

Remarks

You can use the values in the **INumUsedTexInVid** and **INumTexturesUsed** to determine the percentage of textures used during the last frame were resident in video memory (local or non-local). The following code fragment shows how to make this computation:

```
' For this example, the d3dtminfo variable is a D3DDEVINFO_TEXTUREMANAGER  
' structure that has been filled by a call to Direct3DDevice7.GetInfo.  
Dim IHitRate As Long
```

```
' Retrieve the percentage of textures used that were in video memory.  
IHitRate = (d3dtminfo.INumUsedTexInVid * 100) / d3dtminfo.INumTexturesUsed
```

See Also

Direct3DDevice7.GetInfo

D3DDEVINFO_TEXTUREING

The **D3DDEVINFO_TEXTUREING** type contains information about the texturing activity of the application. This structure is used with the **Direct3DDevice7.GetInfo** method.

```
Type D3DDEVINFO_TEXTUREING
  IApproxBytesLoaded As Long
  INumCreates As Long
  INumDestroys As Long
  INumGetDCs As Long
  INumLoads As Long
  INumLocks As Long
  INumPreLoads As Long
  INumSet As Long
  INumSetLODs As Long
  INumSetPriorities As Long
End Type
```

Members

IApproxBytesLoaded

Approximate number bytes loaded by calls to the **Direct3DDevice7.Load** method.

INumCreates

Number of texture surfaces created by the application.

INumDestroys

Number of textures destroyed (released) by the application.

INumGetDCs

Number of times a device context for a texture surface has been retrieved by calling the **DirectDrawSurface7.GetDC** method.

INumLoads

Number of times a texture has been loaded by calling the **Direct3DDevice7.Load** method.

INumLocks

Number of times a texture surface has been locked by calling the **DirectDrawSurface7.Lock** method.

INumPreLoads

Number of times managed textures have been explicitly loaded by calling the **Direct3DDevice7.PreLoad** method.

INumSet

Number of times textures have been set to texture blending stages by calling the **Direct3DDevice7.SetTexture** method.

IDH_D3DDEVINFO_TEXTUREING_d3d_vb

INumSetLODs

Number of times the maximum mipmap level-of-detail has been set by calling the **DirectDrawSurface7.SetLOD** method.

INumSetPriorities

Number of times texture-management priority has been set by calling the **DirectDrawSurface7.SetPriority** method.

See Also

Direct3DDevice7.GetInfo

D3DLIGHT7

The **D3DLIGHT7** type defines the light type in calls to methods such as **Direct3DDevice7.SetLight** and **Direct3DDevice7.GetLight**.

Type D3DLIGHT7

ambient As D3DCOLORVALUE
 attenuation0 As Single
 attenuation1 As Single
 attenuation2 As Single
 diffuse As D3DCOLORVALUE
 direction As D3DVECTOR
 dltType As CONST_D3DLIGHTTYPE
 falloff As Single
 phi As Single
 position As D3DVECTOR
 range As Single
 specular As D3DCOLORVALUE
 theta As Single

End Type

Members**ambient**

Ambient color emitted by the light. This member is a **D3DCOLORVALUE** type.

attenuation0, attenuation1, and attenuation2

Values specifying how a light's intensity changes over distance. (Attenuation does not affect directional lights.) For information about how these attenuation values affect lighting in a scene, see Light Attenuation Over Distance. Valid values for these members range from 0.0 to infinity, inclusive.

diffuse

Diffuse color emitted by the light. This member is a **D3DCOLORVALUE** type.

IDH_D3DLIGHT7_d3d_vb

direction

Direction the light is pointing in world space. This member only has meaning for directional and spotlights. This vector need not be normalized but it should have a nonzero length.

dltType

Type of the light source. This value is one of the following members of the **CONST_D3DLIGHTTYPE** enumeration.

falloff

Decrease in illumination between a spotlight's inner cone (the angle specified by the **theta** member) and the outer edge of the outer cone (the angle specified by the **phi** member). For details on how **falloff** values affect a spotlight, see Spotlight Falloff Model.

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. 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 **D3DLIGHT_RANGE_MAX**, which is defined as the square root of **FLT_MAX**. This member does not affect directional lights.

specular

Specular color emitted by the light. This member is a **D3DCOLORVALUE** type.

theta

Angle, in radians, of the spotlight's inner cone — that is, the fully illuminated spotlight cone. This value must be between 0 and the value specified by the **phi** member.

Remarks

For more information about lights, see Lights.

D3DLIGHTINGCAPS

The **D3DLIGHTINGCAPS** type describes the lighting capabilities of a device. This type is a member of the **D3DDEVICEDESC7** type.

Type D3DLIGHTINGCAPS

IDH_D3DLIGHTINGCAPS_d3d_vb

```

ICaps As CONST_D3DLIGHTCAPSFLAGS
ILightingModel As CONST_D3DLIGHTINGMODELFLAGS
INumLights As Long
End Type

```

Members

ICaps

Flags describing the capabilities of the lighting module. The following constants of the **CONST_D3DLIGHTCAPSFLAGS** enumeration are defined:

D3DLIGHTCAPS_DIRECTIONAL

Supports directional lights.

D3DLIGHTCAPS_POINT

Supports point lights.

D3DLIGHTCAPS_SPOT

Supports spotlights.

ILightingModel

Flags defining whether the lighting model is RGB or monochrome. The following constants of the **CONST_D3DLIGHTINGMODELFLAGS** enumeration are defined:

D3DLIGHTINGMODEL_MONO

Monochromatic lighting model.

D3DLIGHTINGMODEL_RGB

RGB lighting model.

INumLights

Number of lights that can be handled.

D3DLINEPATTERN

The **D3DLINEPATTERN** type describes a line pattern. These values are used by the **D3DRENDERSTATE_LINEPATTERN** render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

Type **D3DLINEPATTERN**

linePattern Long

repeatFactor As Long

End Type

Members

linePattern

Bits specifying the line pattern. For example, the following binary value would produce a dotted line: 1100110011001100.

repeatFactor

IDH_D3DLINEPATTERN_d3d_vb

Number of times to duplicate each series of 1s and 0s specified in the **linePattern** member. This repeat factor allows an application to stretch the line pattern.

Remarks

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**.

D3DLVERTEX

The **D3DLVERTEX** type defines an untransformed and lit vertex (model coordinates with color). An application should use this type when the vertex transformations will be handled by Direct3D. This type contains only data and a color that would be filled by software lighting.

Type D3DLVERTEX

color As Long

specular As Long

tu As Single

tv As Single

x As Single

y As Single

z As Single

End Type

Members

color and specular

Values specifying the color and specular component of the vertex.

tu and tv

Values specifying the texture coordinates of the vertex.

x, y, and z

Values specifying the model coordinates of the vertex.

See Also

D3DTLVERTEX, **D3DVERTEX**

IDH_D3DLVERTEX_d3d_vb

D3DMATERIAL7

The **D3DMATERIAL7** type specifies material properties in calls to the **Direct3DDevice7.GetMaterial** and **Direct3DDevice7.SetMaterial** methods.

```
Type D3DMATERIAL7
    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.

Remarks

To turn off specular highlights for a material, you must set the **power** member to 0—simply setting the specular color components to 0 is not enough.

See Also

Direct3DDevice7.GetMaterial, **Direct3DDevice7.SetMaterial**

D3DMATRIX

The **D3DMATRIX** type describes a matrix for such methods as **Direct3DDevice7.MultiplyTransform**, **Direct3DDevice7.GetTransform**, and **Direct3DDevice7.SetTransform**.

```
Type D3DMATRIX
    rc11 As Single
    rc12 As Single
    rc13 As Single
    rc14 As Single
    rc21 As Single
    rc22 As Single
```

IDH_D3DMATERIAL7_d3d_vb

IDH_D3DMATRIX_d3d_vb

```

rc23 As Single
rc24 As Single
rc31 As Single
rc32 As Single
rc33 As Single
rc34 As Single
rc41 As Single
rc42 As Single
rc43 As Single
rc44 As Single
End Type

```

Remarks

In Direct3D, the **rc34** 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.

See Also

Direct3DDevice7.MultiplyTransform, **Direct3DDevice7.GetTransform**, and **Direct3DDevice7.SetTransform**

D3DPRIMCAPS

The **D3DPRIMCAPS** type defines the capabilities for each primitive type. This type is used when creating a device and when querying the capabilities of a device. This type defines several members in the **D3DDEVICEDESC7** type.

Type D3DPrimCaps

```

AlphaCmpCaps As CONST_D3DCAPSCMP
IDestBlendCaps As CONST_D3DCAPSBLEND
IMiscCaps As CONST_D3DCAPSMISC
IRasterCaps As CONST_D3DCAPSRASTER
IReserved1 As Long
IShadeCaps As CONST_D3DCAPSSHADE
ISrcBlendCaps As CONST_D3DCAPSBLEND
IStippleHeight As Long
IStippleWidth As Long
ITextureAddressCaps As CONST_D3DCAPSTEXTUREADDRESS
ITextureCaps As CONST_D3DCAPSTEXTURE
ITextureFilterCaps As CONST_D3DCAPSTEXTUREFILTER
IZCmpCaps As CONST_D3DCAPSCMP
End Type

```

IDH_D3DPRIMCAPS_d3d_vb

Members

lAlphaCmpCaps

Alpha-test comparison functions that the driver can perform. 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. This member can be one or of the constants of the **CONST_D3DCAPSCMP** enumeration.

lDestBlendCaps

Combination of values from the **CONST_D3DCAPSBLEND** enumeration describing the destination blending capabilities.

lMiscCaps

General capabilities for this primitive. This member can be one or more of the values from the **CONST_D3DCAPSMISC** enumeration.

lRasterCaps

Information on raster-drawing capabilities. This member can be one or more of the constants from the **CONST_D3DCAPSRASTER** enumeration.

lReserved1

Reserved. Do not use.

lShadeCaps

Shading operations that the device can perform. It is assumed, in general, that if a device supports a given command (such as **D3DOP_TRIANGLE**) 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 and Phong shading and whether alpha color components are supported for each of the three color-generation modes. When alpha components are not supported in a given mode, the alpha value of colors generated in that mode 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. These are modified by the shade mode, color model, and by whether the alpha component of a color is blended or stippled. For more information, see 3-D Primitives.

This member can be one or more of the constants of the **CONST_D3DCAPSSHADE** enumeration.

lSrcBlendCaps

Source blending capabilities. This member can be a combination of values from the **CONST_D3DCAPSBLEND** enumeration describing the destination blending capabilities.

lStippleWidth and **lStippleHeight**

Maximum width and height of the supported stipple (up to 32×32).

ITextureAddressCaps

Texture-addressing capabilities. This member can be one or more of the constants of the **CONST_D3DCAPSTEXTUREADDRESS** enumeration.

ITextureCaps

Miscellaneous texture-mapping capabilities. This member can be one or more of the constants of the **CONST_D3DCAPSTEXTURE** enumeration.

ITextureFilterCaps

Texture-map filtering capabilities. 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. This member can any combination of the general and per-stage texture filtering constants of the **CONST_D3DCAPSTEXTUREFILTER** enumeration.

IZCmpCaps

Z-buffer comparison functions that the driver can perform. This member can include same constants of the **CONST_D3DCAPSCMP** as defined for the **IAlphaCmpCaps** member.

D3DRECT

The **D3DRECT** type is a rectangle definition.

```
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

Direct3DDevice7.Clear

IDH_D3DRECT_d3d_vb

D3DTLVERTEX

The **D3DTLVERTEX** type 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 coordinate (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 **sz** 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.

See Also

D3DLVERTEX, **D3DVERTEX**

D3DVECTOR

The **D3DVECTOR** type defines a vector for many Direct3D methods and types.

Type D3DVECTOR

x As Single

y As Single

z As Single

IDH_D3DTLVERTEX_d3d_vb

IDH_D3DVECTOR_d3d_vb

End Type

Members

x, **y**, and **z**

Values describing the vector.

See Also

D3DLIGHT7

D3DVERTEX

The **D3DVERTEX** type 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

Members

x, **y**, and **z**

Values describing the homogeneous coordinates of the vertex.

nx, **ny**, and **nz**

Values describing the normal coordinates of the vertex.

tu and **tv**

Values describing the texture coordinates of the vertex.

See Also

D3DLVERTEX, **D3DTLVERTEX**

IDH_D3DVERTEX_d3d_vb

D3DVERTEXBUFFERDESC

The **D3DVERTEXBUFFERDESC** type describes the properties of a vertex buffer object. This type is used with the **Direct3D7.CreateVertexBuffer** and **Direct3DVertexBuffer7.GetVertexBufferDesc** methods.

Type D3DVERTEXBUFFERDESC
ICaps As CONST_D3DVBCAPSFLAGS
IFVF As CONST_D3DFVFFLAGS
INumVertices As Long
End Type

Members

ICaps

Capability flags that describe the vertex buffer and identify if the vertex buffer can contain optimized vertex data. This parameter can be any combination of the constants of the **CONST_D3DVBCAPSFLAGS** enumeration.

IFVF

A combination of constants of the **CONST_D3DFVFFLAGS** enumeration that describes the vertex format of the vertices in this buffer.

INumVertices

The maximum number of vertices that this vertex buffer can contain.

Remarks

Software devices—MMX and RGB devices—cannot render from a video memory (local or non-local) vertex buffer. To render a vertex buffer using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

See Also

Vertex Buffer Descriptions, Vertex Buffers

D3DVIEWPORT7

The **D3DVIEWPORT7** type defines the window dimensions of a render target surface that a 3-D volume projects onto. This type is used with the **Direct3DDevice7.GetViewport** and **Direct3DDevice7.SetViewport** methods.

Type D3DVIEWPORT7
IHeight As Long

IDH_D3DVERTEXBUFFERDESC_d3d_vb

IDH_D3DVIEWPORT7_d3d_vb

IWidth As Long
IX As Long
IY As Long
maxz As Single
minz As Single
End Type

Members

IHeight and **IWidth**

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.

IX and **IY**

Pixel coordinates of the top-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 zero.

minz and **maxz**

Values describing the range of depth values into which a scene will be rendered. (Note that depth-based clipping in DirectX 7.0 is performed through the values in the projection matrix.) Most applications set these values to 0.0 and 1.0, respectively, but you might set them both to 0.0 to force the system to render objects to the foreground of a scene. Clipping in DirectX 7.0 is performed after applying the projection matrix. For more information, see *Clipping Volumes and The Projection Transformation*.

Remarks

The **IX**, **IY**, **IHeight** and **IWidth** members describe the position and dimensions of the viewport on the render target surface. Usually, applications render to the entire target surface; when rendering to a 640x480 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 ranges 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 (by calling **Direct3DDevice7.SetViewport**), the driver builds a new transformation matrix.

See Also

Direct3DDevice7.GetViewport, **Direct3DDevice7.SetViewport**, *Clipping Volumes, Viewports and Clipping*

DXDRIVERINFO

The **DXDRIVERINFO** type is used in the enumeration methods for DirectDraw, DirectSound and Direct3D to hold driver information.

```
Type DXDRIVERINFO
    strDescription As String
    strGuid As String
    strName As String
End Type
```

Members

strDescription

The textual description of the Direct3D device.

strGuid

The GUID that identifies the Direct3D driver being enumerated.

strName

The name of the Direct3D driver corresponding to this device.

Remarks

This type is also used in DirectDraw and DirectSound.

Enumerations

This section contains information about the following enumerations used with Direct3D Immediate Mode.

- **CONST_D3D**
- **CONST_D3DANTIALIASMODE**
- **CONST_D3DBLEND**
- **CONST_D3DCAPSBLEND**
- **CONST_D3DCAPSCMP**
- **CONST_D3DCAPSMISC**
- **CONST_D3DCAPSRASTER**
- **CONST_D3DCAPSSHADE**
- **CONST_D3DCAPSTEXTURE**
- **CONST_D3DCAPSTEXTUREADDRESS**
- **CONST_D3DCAPSTEXTUREFILTER**
- **CONST_D3DCLEARFLAGS**

IDH_DXDRIVERINFO_d3d_vb

-
- **CONST_D3DCLIPFLAGS**
 - **CONST_D3DCLIPPLANEFLAGS**
 - **CONST_D3DCLIPSTATUSFLAGS**
 - **CONST_D3DCMPFUNC**
 - **CONST_D3DCOLORMODEL**
 - **CONST_D3DCULL**
 - **CONST_D3DDEVICEDESCCAPS**
 - **CONST_D3DDEVICEDESCFLAGS**
 - **CONST_D3DDEVINFOID**
 - **CONST_D3DDPFLAGS**
 - **CONST_D3DFILLMODE**
 - **CONST_D3DFOGMODE**
 - **CONST_D3DFVFCAPSFLAGS**
 - **CONST_D3DFVFFLAGS**
 - **CONST_D3DIMERR**
 - **CONST_D3DLIGHTCAPSFLAGS**
 - **CONST_D3DLIGHTINGMODELFLAGS**
 - **CONST_D3DLIGHTTYPE**
 - **CONST_D3DMATERIALCOLORSOURCE**
 - **CONST_D3DPRIMITIVETYPE**
 - **CONST_D3DPROCESSVERTICESFLAGS**
 - **CONST_D3DRENDERSTATESINGLE**
 - **CONST_D3DRENDERSTATETYPE**
 - **CONST_D3DSHADEMODE**
 - **CONST_D3DSTATEBLOCKTYPE**
 - **CONST_D3DSTENCILCAPSFLAGS**
 - **CONST_D3DSTENCILOP**
 - **CONST_D3DTAFLAGS**
 - **CONST_D3DTEXCOORDINDEXFLAGS**
 - **CONST_D3DTEXOPCAPSFLAGS**
 - **CONST_D3DTEXTUREADDRESS**
 - **CONST_D3DTEXTUREMAGFILTER**
 - **CONST_D3DTEXTUREMINFILTER**
 - **CONST_D3DTEXTUREMIPFILTER**
 - **CONST_D3DTEXTUREOP**
 - **CONST_D3DTEXTURESTAGESINGLE**
 - **CONST_D3DTEXTURESTAGESTATETYPE**

- **CONST_D3DTEXTURETRANSFORMFLAGS**
- **CONST_D3DTRANSFORMSTATETYPE**
- **CONST_D3DVBCAPSFLAGS**
- **CONST_D3DVERTEXBLEND_FLAGS**
- **CONST_D3DVOP_FLAGS**
- **CONST_D3DVTXPCAPS**
- **CONST_D3DZBUFFERTYPE**

CONST_D3D

The **CONST_D3D** enumeration defines miscellaneous constants.

```
Enum CONST_D3D
    D3DDP_MAXTEXCOORD = 8
    D3DRENDERSTATE_WRAPBIAS = 128
    D3DWRAPCOORD_0 = 1
    D3DWRAPCOORD_1 = 2
    D3DWRAPCOORD_2 = 4
    D3DWRAPCOORD_3 = 8
End Enum
```

D3DDP_MAXTEXCOORD

The maximum number of texture coordinates allowed for a vertex.

D3DRENDERSTATE_WRAPBIAS

A convenience value that can be added to a zero-based index for a texture stage to produce a valid **D3DRENDERSTATE_WRAP_n** value for use with the **Direct3DDevice7.SetRenderState** and **Direct3DDevice7.SetRenderState** methods.

D3DWRAPCOORD_0 through D3DWRAPCOORD_3

These cause the system to wrap in the direction of the first, second, third, and fourth dimensions (sometimes referred to as the "s, t, r, and q" directions) for a given texture. These values are used with the **D3DRENDERSTATE_WRAP0** through **D3DRENDERSTATE_WRAP7** render states.

CONST_D3DANTIALIASMODE

The **CONST_D3DANTIALIASMODE** enumeration defines the supported antialiasing mode for the **D3DRENDERSTATE_ANTI_ALIAS** value in the **CONST_D3DRENDERSTATETYPE** enumeration. These values define the settings for antialiasing the edges of primitives. (For more information, see Antialiasing.)

```
Enum CONST_D3DANTIALIASMODE
```

```
# IDH_CONST_D3D_d3d_vb
```

```
# IDH_CONST_D3DANTIALIASMODE_d3d_vb
```

```

D3DANTIALIAS_NONE      = 0
D3DANTIALIAS_SORTDEPENDENT = 1
D3DANTIALIAS_SORTINDEPENDENT = 2
End Enum

```

D3DANTIALIAS_NONE

No antialiasing is performed. This is the default setting.

D3DANTIALIAS_SORTDEPENDENT

Antialiasing is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur.

D3DANTIALIAS_SORTINDEPENDENT

Antialiasing is not dependent on the sort order of the polygons.

CONST_D3DBLEND

The **CONST_D3DBLEND** enumeration defines the supported blend mode for the **D3DRENDERSTATE_DESTBLEND** values in the **CONST_D3DRENDERSTATETYPE** enumeration. In the member descriptions that follow, the RGBA values of the source and destination are indicated with the subscripts *s* and *d*.

```

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

```

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).

IDH_CONST_D3DBLEND_d3d_vb

D3DBLEND_INVSRCOLORBlend 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_BOTHINVSRCALPHASource 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**

D3DBLEND_BOTHSRCALPHA is no longer supported in DirectX 6.0 and later. Please explicitly set both D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA separately.

CONST_D3DCAPSBLEND

The **CONST_D3DCAPSBLEND** enumeration defines the blending capabilities for a device. These values are combined in the **ISrcBlendCaps** and **IDestBlendCaps** members of the **D3DPRIMCAPS** type.

Enum **CONST_D3DCAPSBLEND**

D3DPBLENDCAPS_BOTHINVSRCALPHA = 4096

D3DPBLENDCAPS_BOTHSRCALPHA = 2048

D3DPBLENDCAPS_DESTALPHA = 64

D3DPBLENDCAPS_DESTCOLOR = 256

D3DPBLENDCAPS_INVDESTALPHA = 128

D3DPBLENDCAPS_INVDESTCOLOR = 512

D3DPBLENDCAPS_INVSRCALPHA = 32

D3DPBLENDCAPS_INVSRCOLOR = 8

IDH_CONST_D3DCAPSBLEND_d3d_vb

```

D3DPBLENDCAPS_ONE =2
D3DPBLENDCAPS_SRCALPHA =16
D3DPBLENDCAPS_SRCALPHASAT =1024
D3DPBLENDCAPS_SRCCOLOR =4
D3DPBLENDCAPS_ZERO =1
End Enum

```

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

Source blend factor is (A_s , A_s , A_s , A_s) and destination blend factor is (1- A_s , 1- A_s , 1- A_s , 1- A_s); the destination blend selection is overridden.

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_INVSRCOLOR

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- A_d).

D3DPBLENDCAPS_SRCCOLOR

Blend factor is (R_s , G_s , B_s , A_s).

D3DPBLENDCAPS_ZERO

Blend factor is (0, 0, 0, 0).

CONST_D3DCAPSCMP

The **CONST_D3DCAPSCMP** enumeration defines comparison capabilities for depth-buffer comparisons and alpha-testing. These flags are combined and present in the **IZCmpCaps** and **lAlphaCmpCaps** members of the **D3DPRIMCAPS** type.

```
# IDH_CONST_D3DCAPSCMP_d3d_vb
```

```

Enum CONST_D3DCAPSCMP
    D3DPCMPCAPS_ALWAYS = 128
    D3DPCMPCAPS_EQUAL = 4
    D3DPCMPCAPS_GREATER = 16
    D3DPCMPCAPS_GREATEREQUAL = 64
    D3DPCMPCAPS_LESS = 2
    D3DPCMPCAPS_LESSEQUAL = 8
    D3DPCMPCAPS_NEVER = 1
    D3DPCMPCAPS_NOTEQUAL = 32
End Enum

```

D3DPCMPCAPS_ALWAYS
Always pass the comparison.

D3DPCMPCAPS_EQUAL
Pass the comparison if the new value equals the current value.

D3DPCMPCAPS_GREATER
Pass the comparison if the new value is greater than the current value.

D3DPCMPCAPS_GREATEREQUAL
Pass the comparison if the new value is greater than or equal to the current value.

D3DPCMPCAPS_LESS
Pass the comparison if the new value is less than the current value.

D3DPCMPCAPS_LESSEQUAL
Pass the comparison if the new value is less than or equal to the current value.

D3DPCMPCAPS_NEVER
Always fail the comparison.

D3DPCMPCAPS_NOTEQUAL
Pass the comparison if the new value does not equal the current value.

CONST_D3DCAPSMISC

The **CONST_D3DCAPSMISC** enumeration defines capability flags that are combined and found in the **IMiscCaps** member of the **D3DPRIMCAPS** type.

```

Enum CONST_D3DCAPSMISC
    D3DPMISCCAPS_CONFORMANT = 8
    D3DPMISCCAPS_CULLCCW = 64
    D3DPMISCCAPS_CULLCW = 32
    D3DPMISCCAPS_CULLNONE = 16
    D3DPMISCCAPS_LINEPATTERNREP = 4
    D3DPMISCCAPS_MASKPLANES = 1
    D3DPMISCCAPS_MASKZ = 2
End Enum

```

IDH_CONST_D3DCAPSMISC_d3d_vb

D3DPMISCCAPS_CONFORMANT

The device conforms to the OpenGL standard.

D3DPMISCCAPS_CULLCCW

The driver supports counterclockwise culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CCW constant of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_CULLCW

The driver supports clockwise triangle culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CW constant of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_CULLNONE

The driver does not perform triangle culling. This corresponds to the D3DCULL_NONE constant of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_LINEPATTERNREP

Not used.

D3DPMISCCAPS_MASKPLANES

The device can perform a bitmask of color planes.

D3DPMISCCAPS_MASKZ

The device can enable and disable modification of the depth-buffer on pixel operations.

CONST_D3DCAPSRASTER

The **CONST_D3DCAPSRASTER** enumeration defines rasterization capability flags that are combined and present in the **IRasterCaps** member of the **D3DPRIMCAPS** type.

Enum **CONST_D3DCAPSRASTER**

```

D3DPRASTERCAPS_ANISOTROPY = 131072
D3DPRASTERCAPS_ANTIALIASEDGEDGES = 4096
D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT = 1024
D3DPRASTERCAPS_DITHER = 1
D3DPRASTERCAPS_FOGRANGE = 65536
D3DPRASTERCAPS_FOGTABLE = 256
D3DPRASTERCAPS_FOGVERTEX = 128
D3DPRASTERCAPS_MIPMAPLODBIAS = 8192
D3DPRASTERCAPS_PAT = 8
D3DPRASTERCAPS_ROP2 = 2
D3DPRASTERCAPS_STIPPLE = 512
D3DPRASTERCAPS_SUBPIXEL = 32
D3DPRASTERCAPS_SUBPIXELX = 64
D3DPRASTERCAPS_XOR = 4

```

IDH_CONST_D3DCAPSRASTER_d3d_vb

```
D3DPRASERCAPS_ZBIAS = 16384
D3DPRASERCAPS_ZBUFFERLESSHSR = 32768
D3DPRASERCAPS_ZTEST = 16
```

End Enum

D3DPRASERCAPS_ANISOTROPY

The device supports anisotropic filtering.

D3DPRASERCAPS_ANTIALIASEDGES

The device can antialias lines forming the convex outline of objects.

D3DPRASERCAPS_ANTIALIASSORTDEPENDENT

The device supports antialiasing that is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur. For more information, see the **CONST_D3DANTIALIASMODE** enumeration.

D3DPRASERCAPS_ANTIALIASSORTINDEPENDENT

The device supports antialiasing that is not dependent on the sort order of the polygons. For more information, see the **CONST_D3DANTIALIASMODE** enumeration.

D3DPRASERCAPS_DITHER

The device can dither to improve color resolution.

D3DPRASERCAPS_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.

D3DPRASERCAPS_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.

D3DPRASERCAPS_FOGVERTEX

The device calculates the fog value during the lighting operation, places the value into the alpha component given for the **specular** member of the **D3DTLVERTEX** type, and interpolates the fog value during rasterization.

D3DPRASERCAPS_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**.

D3DPRASERCAPS_PAT

The driver can perform patterned drawing for the primitive being queried.

D3DPRASERCAPS_ROP2

The device can support raster operations other than **R2_COPYPEN**.

D3DPRASERCAPS_STIPPLE

The device can stipple polygons to simulate translucency.

D3DPRASERCAPS_SUBPIXEL

The device performs subpixel placement of z, color, and texture data, rather than working with the nearest integer pixel coordinate. This helps avoid bleed-through due to z imprecision, and jitter of color and texture values for pixels. Note that there is no corresponding state that can be enabled and disabled; the device either performs subpixel placement or it does not, and this bit is present only so that the Direct3D client will be better able to determine what the rendering quality will be.

D3DPRASERCAPS_SUBPIXELX

The device is subpixel accurate along the x-axis only and is clamped to an integer y-axis scan line.

D3DPRASERCAPS_WBUFFER

The device supports depth buffering using w. For more information, see Depth Buffers.

D3DPRASERCAPS_WFOG

The device supports w-based fog. W-based fog is used when a perspective projection matrix is specified, but affine projections will 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.

D3DPRASERCAPS_XOR

The device can support **XOR** GDI raster operations. If this flag is not set but D3DPRIM_RASTER_ROP2 is set, then **XOR** operations must still be supported.

D3DPRASERCAPS_ZBIAS

The device supports z-bias values. These are integer values assigned to polygons that allow physically coplanar polygons to appear separate. For more information, see D3DRENDERSTATE_ZBIAS in the **CONST_D3DRENDERSTATETYPE** enumeration.

D3DPRASERCAPS_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 hidden-surface removal is hardware-dependent and is transparent to the application.

Z-bufferless HSR is performed if no depth-buffer surface is attached to the rendering-target surface and the depth-buffer comparison test is enabled (that is, when the state value associated with the D3DRENDERSTATE_ZENABLE enumeration constant is set to True).

D3DPRASERCAPS_ZTEST

The device can perform z-test operations. This effectively renders a primitive and indicates whether any z pixels would have been rendered.

CONST_D3DCAPSSHADE

The **CONST_D3DCAPSSHADE** enumeration defines polygon shading capability flags that are combined and present in the **IShadeCaps** member of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSSHADE
    D3DPSHADECAPS_ALPHAFLATBLEND = 4096
    D3DPSHADECAPS_ALPHAFLATSTIPPLED = 8192
    D3DPSHADECAPS_ALPHAGOURAUBLEND = 16384
    D3DPSHADECAPS_ALPHAGOURAUDSTIPPLED = 32768
    D3DPSHADECAPS_ALPHAPHONGBLEND = 65536
    D3DPSHADECAPS_ALPHAPHONGSTIPPLED = 131072
    D3DPSHADECAPS_COLORFLATMONO = 1
    D3DPSHADECAPS_COLORFLATRGB = 2
    D3DPSHADECAPS_COLORGOURAUDMONO = 4
    D3DPSHADECAPS_COLORGOURAUDRGB = 8
    D3DPSHADECAPS_COLORPHONGMONO = 16
    D3DPSHADECAPS_COLORPHONGRGB = 32
    D3DPSHADECAPS_FOGFLAT = 262144
    D3DPSHADECAPS_FOGGOURAUD = 524288
    D3DPSHADECAPS_FOGPHONG = 1048576
    D3DPSHADECAPS_SPECULARFLATMONO = 64
    D3DPSHADECAPS_SPECULARFLATRGB = 128
    D3DPSHADECAPS_SPECULARGOURAUDMONO = 256
    D3DPSHADECAPS_SPECULARGOURAUDRGB = 512
    D3DPSHADECAPS_SPECULARPHONGMONO = 1024
    D3DPSHADECAPS_SPECULARPHONGRGB = 2048
End Enum
```

D3DPSHADECAPS_ALPHAFLATBLEND,
D3DPSHADECAPS_ALPHAFLATSTIPPLED

Device can support an alpha component for flat blended and stippled transparency, respectively (the D3DSHADE_FLAT state for the **CONST_D3DSHADEMODE** enumeration). In these modes, the alpha color component for a primitive is provided as part of the color for the first vertex of the primitive.

D3DPSHADECAPS_ALPHAGOURAUBLEND,
D3DPSHADECAPS_ALPHAGOURAUDSTIPPLED

Device can support an alpha component for Gouraud blended and stippled transparency, respectively (the D3DSHADE_GOURAUD state for the **CONST_D3DSHADEMODE** enumeration). In these modes, the alpha color component for a primitive is provided at vertices and interpolated across a face along with the other color components.

IDH_CONST_D3DCAPSSHADE_d3d_vb

D3DPSHADECAPS_ALPHAPHONGBLEND,
D3DPSHADECAPS_ALPHAPHONGSTIPPLED

Device can support an alpha component for Phong blended and stippled transparency, respectively (the D3DSHADE_PHONG state for the **CONST_D3DSHADEMODE** enumeration). In these modes, vertex parameters are reevaluated on a per-pixel basis, applying lighting effects for the red, green, and blue color components. Phong shading is not currently supported.

D3DPSHADECAPS_COLORFLATMONO,
D3DPSHADECAPS_COLORFLATRGB

Device can support colored flat shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, the color component for a primitive is provided as part of the color for the first vertex of the primitive. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORGOURAUDMONO,
D3DPSHADECAPS_COLORGOURAURGB

Device can support colored Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, the color component for a primitive is provided at vertices and interpolated across a face along with the other color components. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORPHONGMONO,
D3DPSHADECAPS_COLORPHONGRGB

Device can support colored Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, vertex parameters are reevaluated on a per-pixel basis. Lighting effects are applied for the red, green, and blue color components in RGB mode, and for the blue component only for monochromatic mode. Phong shading is not currently supported.

D3DPSHADECAPS_FOGFLAT, D3DPSHADECAPS_FOGGOURAUD,
D3DPSHADECAPS_FOGPHONG

Device can support fog in the flat, Gouraud, and Phong shading models, respectively. Phong shading is not currently supported.

D3DPSHADECAPS_SPECULARFLATMONO,
D3DPSHADECAPS_SPECULARFLATRGB

Device can support specular highlights in flat shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARGOURAUDMONO,
D3DPSHADECAPS_SPECULARGOURAURGB

Device can support specular highlights in Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARPHONGMONO,
D3DPSHADECAPS_SPECULARPHONGRGB

Device can support specular highlights in Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. Phong shading is not currently supported.

CONST_D3DCAPSTEXTURE

The **CONST_D3DCAPSTEXTURE** enumeration defines texturing capability flags that are combined and present in the **ITextureCaps** member of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSTEXTURE {
    D3DPTEXTURECAPS_ALPHA = 4
    D3DPTEXTURECAPS_BORDER = 16
    D3DPTEXTURECAPS_COLORKEYBLEND = 4096
    D3DPTEXTURECAPS_CUBEMAP = 1024
    D3DPTEXTURECAPS_NONPOW2CONDITIONAL = 256
    D3DPTEXTURECAPS_PERSPECTIVE = 1
    D3DPTEXTURECAPS_POW2 = 2
    D3DPTEXTURECAPS_SQUAREONLY = 32
    D3DPTEXTURECAPS_TRANSPARENCY = 8
End Enum
```

D3DPTEXTURECAPS_ALPHA

Supports RGBA textures in the **D3DTEX_DECAL** and **D3DTEX_MODULATE** texture filtering modes. If this capability is not set, then only RGB textures are supported in those modes. Regardless of the setting of this flag, alpha must always be supported in **D3DTEX_DECAL_MASK**, **D3DTEX_DECAL_ALPHA**, and **D3DTEX_MODULATE_ALPHA** filtering modes whenever those filtering modes are available.

D3DPTEXTURECAPS_ALPHAPALETTE

Supports palettized texture surfaces whose palettes contain alpha information (see **DDPCAPS_ALPHA** in the **DDCAPS** type).

D3DPTEXTURECAPS_COLORKEYBLEND

The device supports alpha-blended colorkeying through the use of the **D3DRENDERSTATE_COLORKEYBLENDENABLE** render state.

D3DPTEXTURECAPS_CUBEMAP

Supports cubic environment mapping.

D3DPTEXTURECAPS_BORDER

Superseded by **D3DPTADDRESSCAPS_BORDER**.

D3DPTEXTURECAPS_PERSPECTIVE

Perspective correction is supported.

D3DPTEXTURECAPS_PROJECTED

IDH_CONST_D3DCAPSTEXTURE_d3d_vb

Supports the D3DTTFF_PROJECTED texture transformation flag. When applied, the device divides transformed texture coordinates by the last texture coordinate.

D3DPTEXTURECAPS_NONPOW2CONDITIONAL

Conditionally supports the use of textures with dimensions that are not powers of two. 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 D3DTEXTADDRESS_CLAMP.
- Texture wrapping for the texture stage is disabled (D3DRENDERSTATE_WRAP n set to zero).
- Mipmapping is not in use. (Mipmapped textures must have dimensions that are powers of two.)

D3DPTEXTURECAPS_POW2

All nonmipmapped textures must have widths and heights specified as powers of two if this flag is set. (Note that all mipmapped textures must always have dimensions that are powers of two.)

D3DPTEXTURECAPS_SQUAREONLY

All textures must be square.

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDDBYSIZE

Texture indices are not scaled by the texture size prior to interpolation.

D3DPTEXTURECAPS_TRANSPARENCY

Texture transparency is supported. (Only those texels that are not the current transparent color are drawn.)

CONST_D3DCAPSTEXTUREADDRESS

The **CONST_D3DCAPSTEXTUREADDRESS** enumeration defines texture addressing capability flags that are combined and present in the **ITextureAddressCaps** member of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSTEXTUREADDRESS
    D3DPTADDRESSCAPS_BORDER = 8
    D3DPTADDRESSCAPS_CLAMP = 4
    D3DPTADDRESSCAPS_INDEPENDENTUV = 16
    D3DPTADDRESSCAPS_MIRROR = 2
    D3DPTADDRESSCAPS_WRAP = 1
End Enum
```

D3DPTADDRESSCAPS_BORDER

IDH_CONST_D3DCAPSTEXTUREADDRESS_d3d_vb

Device supports setting coordinates outside the range [0.0, 1.0] to the border color, as specified by the D3DTSS_BORDERCOLOR texture stage state. This ability corresponds to the **D3DTADDRESS_BORDER** texture-addressing mode.

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_WRAP

Device can wrap textures to addresses.

CONST_D3DCAPSTEXTUREFILTER

The **CONST_D3DCAPSTEXTUREFILTER** enumeration defines texture filtering capability flags that are combined and present in the **ITextureFilterCaps** of the **D3DPRIMCAPS** type.

Enum CONST_D3DCAPSTEXTUREFILTER

```
D3DPTFILTERCAPS_LINEAR = 2
D3DPTFILTERCAPS_LINEARMIPLINEAR = 32
D3DPTFILTERCAPS_LINEARMIPNEAREST = 16
D3DPTFILTERCAPS_MAGFAFLATCUBIC = 134217728
D3DPTFILTERCAPS_MAGFANISOTROPIC = 67108864
D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC = 268435456
D3DPTFILTERCAPS_MAGFLINEAR = 16777216
D3DPTFILTERCAPS_MAGFPOINT = 8388608
D3DPTFILTERCAPS_MINFANISOTROPIC = 1024
D3DPTFILTERCAPS_MINFLINEAR = 512
D3DPTFILTERCAPS_MINFPOINT = 256
D3DPTFILTERCAPS_MIPFLINEAR = 131072
D3DPTFILTERCAPS_MIPFPOINT = 65536
D3DPTFILTERCAPS_MIPLINEAR = 8
D3DPTFILTERCAPS_MIPNEAREST = 4
D3DPTFILTERCAPS_NEAREST = 1
```

End Enum

General texture filtering flags

D3DPTFILTERCAPS_LINEAR

IDH_CONST_D3DCAPSTEXTUREFILTER_d3d_vb

Bilinear filtering. Chooses the texel that has nearest coordinates, then performs a weighted average with the four surrounding texels to determine the final color. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DPTFILTERCAPS_LINEARMIPLINEAR

Trilinear interpolation between mipmaps. Performs bilinear filtering on the two nearest mipmaps, then interpolates linearly between the two colors to determine a final color.

D3DPTFILTERCAPS_LINEARMIPNEAREST

Linear interpolation between two point sampled mipmaps. Chooses the nearest texel from the two closest mipmap levels, then performs linear interpolation between them.

D3DPTFILTERCAPS_MIPLINEAR

Nearest mipmapping, with bilinear filtering applied to the result. Chooses the texel from the appropriate mipmap that has nearest coordinates, then performs a weighted average with the four surrounding texels to determine the final color.

D3DPTFILTERCAPS_MIPNEAREST

Nearest mipmapping. Chooses the texel from the appropriate mipmap with coordinates nearest to the desired pixel value.

D3DPTFILTERCAPS_NEAREST

Point sampling. The texel with coordinates nearest to the desired pixel value is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

Per-stage texture filtering flags

D3DPTFILTERCAPS_MAGFAFLATCUBIC

The device supports per-stage flat-cubic filtering for magnifying textures. The flat-cubic magnification filter is represented by the D3DTFG_FLATCUBIC member of the **CONST_D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MAGFANISOTROPIC

The device supports per-stage anisotropic filtering for magnifying textures. The anisotropic magnification filter is represented by the D3DTFG_ANISOTROPIC member of the **CONST_D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC

The device supports the per-stage Gaussian-cubic filtering for magnifying textures. The Gaussian-cubic magnification filter is represented by the D3DTFG_GAUSSIANCUBIC member of the **CONST_D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MAGFLINEAR

The device supports per-stage bilinear-interpolation filtering for magnifying textures. The bilinear-interpolation magnification filter is represented by the D3DTFG_LINEAR member of the **CONST_D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MAGFPOINT

The device supports per-stage point-sampled filtering for magnifying textures. The point-sample magnification filter is represented by the D3DTFG_POINT member of the **CONST_D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MINFANISOTROPIC

The device supports per-stage anisotropic filtering for minifying textures. The anisotropic minification filter is represented by the D3DTFN_ANISOTROPIC member of the **CONST_D3DTEXTUREMINFILTER** enumeration.

D3DPTFILTERCAPS_MINFLINEAR

The device supports per-stage bilinear-interpolation filtering for minifying textures. The bilinear minification filter is represented by the D3DTFN_LINEAR member of the **CONST_D3DTEXTUREMINFILTER** enumeration.

D3DPTFILTERCAPS_MINFPOINT

The device supports per-stage point-sampled filtering for minifying textures. The point-sample minification filter is represented by the D3DTFN_POINT member of the **CONST_D3DTEXTUREMINFILTER** enumeration.

D3DPTFILTERCAPS_MIPFLINEAR

The device supports per-stage trilinear-interpolation filtering for mipmaps. The trilinear-interpolation mipmapping filter is represented by the D3DTFP_LINEAR member of the **CONST_D3DTEXTUREMIPFILTER** enumeration.

D3DPTFILTERCAPS_MIPFPOINT

The device supports per-stage point-sampled filtering for mipmaps. The point-sample mipmapping filter is represented by the D3DTFP_POINT member of the **CONST_D3DTEXTUREMIPFILTER** enumeration.

CONST_D3DCLEARFLAGS

The **CONST_D3DCLEARFLAGS** enumeration defines flags that are used to determine the behavior of the **Direct3DDevice7.Clear** method.

Enum **CONST_D3DCLEARFLAGS**

D3DCLEAR_ALL = 7

D3DCLEAR_STENCIL = 4

D3DCLEAR_TARGET = 1

D3DCLEAR_ZBUFFER = 2

End Enum

D3DCLEAR_ALL

Clear the rendering target, stencil buffer, and depth-buffer surfaces.

D3DCLEAR_STENCIL

Clear the stencil buffer to the value in the *stencil* parameter.

D3DCLEAR_TARGET

Clear the rendering target to the color in the *color* parameter.

IDH_CONST_D3DCLEARFLAGS_d3d_vb

D3DCLEAR_ZBUFFER

Clear the depth-buffer to the value in the *z* parameter.

CONST_D3DCLIPFLAGS

The **CONST_D3DCLIPFLAGS** enumeration defines clipping flags returned by the **Direct3DDevice7.ComputeSphereVisibility** method.

```
Enum CONST_D3DCLIPFLAGS
    D3DSTATUS_CLIPINTERSECTIONALL = 17891328
    D3DSTATUS_CLIPUNIONALL = 17891328
    D3DSTATUS_DEFAULT = 34668544
    D3DSTATUS_ZNOTVISIBLE = 16777216
    D3DSTATUS_CLIPINTERSECTIONBACK = 131072
    D3DSTATUS_CLIPINTERSECTIONBOTTOM = 32768
    D3DSTATUS_CLIPINTERSECTIONFRONT = 65536
    D3DSTATUS_CLIPINTERSECTIONGEN0 = 262144
    ' D3DSTATUS_CLIPINTERSECTIONGEN1 to
D3DSTATUS_CLIPINTERSECTIONGEN4
    ' omitted here for brevity.
    D3DSTATUS_CLIPINTERSECTIONGEN5 = 8388608
    D3DSTATUS_CLIPINTERSECTIONLEFT = 4096
    D3DSTATUS_CLIPINTERSECTIONRIGHT = 8192
    D3DSTATUS_CLIPINTERSECTIONTOP = 16384
    D3DSTATUS_CLIPUNIONBACK = 32
    D3DSTATUS_CLIPUNIONBOTTOM = 8
    D3DSTATUS_CLIPUNIONFRONT = 16
    D3DSTATUS_CLIPUNIONGEN0 = 64
    ' D3DSTATUS_CLIPUNIONGEN1 to D3DSTATUS_CLIPUNIONGEN4
    ' omitted here for brevity.
    D3DSTATUS_CLIPUNIONGEN5 = 2048
    D3DSTATUS_CLIPUNIONLEFT = 1
    D3DSTATUS_CLIPUNIONRIGHT = 2
    D3DSTATUS_CLIPUNIONTOP = 4
    D3DCLIP_BACK = 32
    D3DCLIP_BOTTOM = 8
    D3DCLIP_FRONT = 16
    D3DCLIP_GEN0 = 64
    ' D3DCLIP_GEN1 to D3DCLIP_GEN4 omitted here for brevity.
    D3DCLIP_GEN5 = 2048
    D3DCLIP_LEFT = 1
    D3DCLIP_RIGHT = 2
    D3DCLIP_TOP = 4
End Enum
```

IDH_CONST_D3DCLIPFLAGS_d3d_vb

Combination and General Flags

D3DSTATUS_CLIPINTERSECTIONALL

Combination of all CLIPINTERSECTION flags.

D3DSTATUS_CLIPUNIONALL

Combination of all CLIPUNION flags.

D3DSTATUS_DEFAULT

Combination of D3DSTATUS_CLIPINTERSECTIONALL and D3DSTATUS_ZNOTVISIBLE flags. This value is the default.

D3DSTATUS_ZNOTVISIBLE

Indicates that the rendered primitive is not visible. This flag is set or cleared by the system when rendering with z-checking enabled (see D3DRENDERSTATE_ZVISIBLE).

Clip Intersection Flags

D3DSTATUS_CLIPINTERSECTIONBACK

Logical **And** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM

Logical **And** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT

Logical **And** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through

D3DSTATUS_CLIPINTERSECTIONGEN5

Logical **And** of the clip flags for application-defined clipping planes.

D3DSTATUS_CLIPINTERSECTIONLEFT

Logical **And** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT

Logical **And** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP

Logical **And** of the clip flags for the vertices compared to the top of the viewing frustum.

Clip Union Flags

D3DSTATUS_CLIPUNIONBACK

Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM

Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT

Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5

Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT

Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT

Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP

Equal to D3DCLIP_TOP.

Basic Clipping Flags

D3DCLIP_BACK

All vertices are clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM

All vertices are clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT

All vertices are clipped by the front plane of the viewing frustum.

D3DCLIP_LEFT

All vertices are clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT

All vertices are clipped by the right plane of the viewing frustum.

D3DCLIP_TOP

All vertices are clipped by the top plane of the viewing frustum.

D3DCLIP_GEN0 through D3DCLIP_GEN5

All vertices are clipped by a corresponding application-defined clipping plane.

CONST_D3DCLIPPLANEFLAGS

The **CONST_D3DCLIPPLANEFLAGS** enumeration defines flags that can be combined and used with the D3DRENDERSTATE_CLIPPLANEENABLE render state.

Type CONST_D3DCLIPPLANEFLAGS

D3DCPF_DISABLEALL = 0

D3DCPF_ENABLEPLANE0 = 1

,

' D3DCPF_ENABLEPLANE1 to D3DCPF_ENABLEPLANE30 omitted here.

,

D3DCPF_ENABLEPLANE31 = -2147483648

End Type

D3DCPF_DISABLEALL

Disable all clipping planes. This flag should not be combined with other flags.

D3DCPF_ENABLEPLANE0 through D3DCPF_ENABLEPLANE31

IDH_CONST_D3DCLIPPLANEFLAGS_d3d_vb

Enable the clipping plane, or planes, at the corresponding index. These values can be combined to simultaneously enable multiple clipping planes.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DCLIPSTATUSFLAGS

The **CONST_D3DCLIPSTATUSFLAGS** enumeration defines flags that are used in the **D3DCLIPSTATUS** type.

```
Enum CONST_D3DCLIPSTATUSFLAGS
    D3DCLIPSTATUS_EXTENTS2 = 2
    D3DCLIPSTATUS_EXTENTS3 = 4
    D3DCLIPSTATUS_STATUS = 1
End Enum
```

D3DCLIPSTATUS_EXTENTS2

The type describes the current 2-D extents. This flag cannot be combined with D3DCLIPSTATUS_EXTENTS3.

D3DCLIPSTATUS_EXTENTS3

Not currently implemented.

D3DCLIPSTATUS_STATUS

The type describes the current clip status.

CONST_D3DCMPFUNC

The **CONST_D3DCMPFUNC** enumeration defines the supported compare functions for the D3DRENDERSTATE_ZFUNC, D3DRENDERSTATE_ALPHAFUNC, and D3DRENDERSTATE_STENCILFUNC render states.

```
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
```

IDH_CONST_D3DCLIPSTATUSFLAGS_d3d_vb

IDH_CONST_D3DCMPFUNC_d3d_vb

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.

CONST_D3DCOLORMODEL

The **CONST_D3DCOLORMODEL** enumeration is used to define the color model in which the system will run. A driver can expose either or both flags in the **CONST_COLORMODEL** member of the **D3DDEVICEDESC7** type.

Enum **CONST_D3DCOLORMODEL**

D3DCOLOR_MONO = 1

D3DCOLOR_RGB = 2

End Enum

D3DCOLOR_MONO

Use a monochromatic model (or ramp model). In this model, the blue component of a vertex color is used to define the brightness of a lit vertex.

D3DCOLOR_RGB

Use a full RGB model.

IDH_CONST_D3DCOLORMODEL_d3d_vb

CONST_D3DCULL

The **CONST_D3DCULL** enumeration defines the supported cull modes used with the **D3DRENDERSTATE_CULLMODE** render state. These define how back faces are culled when rendering a geometry.

```
Enum CONST_D3DCULL
    D3DCULL_NONE = 1
    D3DCULL_CW  = 2
    D3DCULL_CCW = 3
End Enum
```

D3DCULL_NONE

Do not cull back faces.

D3DCULL_CW

Cull back faces with clockwise vertices.

D3DCULL_CCW

Cull back faces with counterclockwise vertices.

See Also

D3DPRIMCAPS, **CONST_D3DRENDERSTATETYPE**

CONST_D3DDEVICEDESCCAPS

The **CONST_D3DDEVICEDESCCAPS** enumeration defines device capability flags that are combined and present in the **IDevCaps** member of the **D3DDEVICEDESC7** type.

```
Enum CONST_D3DDEVICEDESCCAPS
    D3DDEVCAPS_CANBLTSYSTONONLOCAL = 131072
    D3DDEVCAPS_CANRENDERAFTERFLIP = 2048
    D3DDEVCAPS_DRAWPRIMTLVERTEX = 1024
    D3DDEVCAPS_FLOATTLVERTEX = 1
    D3DDEVCAPS_HWRASTERIZATION = 524288
    D3DDEVCAPS_HWTRANSFORMANDLIGHT = 65536
    D3DDEVCAPS_SEPARATETEXTUREMEMORIES = 16384
    D3DDEVCAPS_SORTDECREASINGZ = 4
    D3DDEVCAPS_SORTEACT = 8
    D3DDEVCAPS_SORTINCREASINGZ = 2
    D3DDEVCAPS_TEXTURENONLOCALVIDMEM = 4096
    D3DDEVCAPS_TEXTURESYSTEMMEMORY = 256
    D3DDEVCAPS_TEXTUREVIDEOMEMORY = 512
```

IDH_CONST_D3DCULL_d3d_vb

IDH_CONST_D3DDEVICEDESCCAPS_d3d_vb

D3DDEVCAPS_TLVERTEXSYSTEMMEMORY = 64

D3DDEVCAPS_TLVERTEXVIDEOMEMORY = 128

End Enum

D3DDEVCAPS_CANBLTSYSTONONLOCAL

Device supports blits from system-memory textures to non-local video-memory textures.

D3DDEVCAPS_CANRENDERAFTERFLIP

Device can queue rendering commands after a page flip. Applications should 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 HAL.

D3DDEVCAPS_FLOATTLVERTEX

Device accepts floating point for post-transform vertex data.

D3DDEVCAPS_HWRASTERIZATION

Device has hardware acceleration for scene rasterization.

D3DDEVCAPS_HWTRANSFORMANDLIGHT

Device supports transformation and lighting in hardware.

D3DDEVCAPS_SEPARATETEXTUREMEMORIES

Device uses discrete texture memory pools for each stage. Textures must be assigned to texture stages explicitly at creation-time by setting the **ITextureStage** member of the **DDSURFACEDESC2** type to the appropriate stage identifier.

D3DDEVCAPS_SORTDECREASINGZ

Device needs data sorted for decreasing depth.

D3DDEVCAPS_SORTEACT

Device needs data sorted exactly.

D3DDEVCAPS_SORTINCREASINGZ

Device needs data sorted for increasing depth.

D3DDEVCAPS_TEXTURENONLOCALVIDMEM

Device can retrieve textures from non-local video (AGP) memory. For more information about AGP memory, see Using Non-local Video Memory Surfaces in the DirectDraw documentation.

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.

CONST_D3DDEVICEDESCFLAGS

The **CONST_D3DDEVICEDESCFLAGS** enumeration defines flags that are used in the **IFlags** member of the **D3DDEVICEDESC7** type.

```
Enum CONST_D3DDEVICEDESCFLAGS
    D3DDD_BCLIPPING = 16
    D3DDD_DEVCAPS = 2
    D3DDD_DEVICERENDERBITDEPTH = 128
    D3DDD_DEVICEZBUFFERBITDEPTH = 256
    D3DDD_LIGHTINGCAPS = 8
    D3DDD_LINECAPS = 32
    D3DDD_MAXBUFFERSIZE = 512
    D3DDD_MAXVERTEXCOUNT = 1024
    D3DDD_TRANSFORMCAPS = 4
    D3DDD_TRICAPS = 64
End Enum
```

D3DDD_BCLIPPING

The **IClipping** member of the **D3DDEVICEDESC7** type is valid.

D3DDD_DEVCAPS

The **IDevCaps** member of the **D3DDEVICEDESC7** type is valid.

D3DDD_DEVICERENDERBITDEPTH

The **IDeviceRenderBitDepth** member of the **D3DDEVICEDESC7** type is valid.

D3DDD_DEVICEZBUFFERBITDEPTH

The **IDeviceZBufferBitDepth** member of the **D3DDEVICEDESC7** type is valid.

D3DDD_LIGHTINGCAPS

The **dlcLightingCaps** member of the **D3DDEVICEDESC7** type is valid.

D3DDD_LINECAPS

The **dpcLineCaps** member of the **D3DDEVICEDESC7** type is valid.

D3DDD_MAXBUFFERSIZE

The **IMaxBufferSize** member of the **D3DDEVICEDESC7** type is valid.

D3DDD_MAXVERTEXCOUNT

The **IMaxVertexCount** member of the **D3DDEVICEDESC7** type is valid.

D3DDD_TRANSFORMCAPS

The **ITransformCaps** member of the **D3DDEVICEDESC7** type is valid.

D3DDD_TRICAPS

The **dpcTriCaps** member of the **D3DDEVICEDESC7** type is valid.

IDH_CONST_D3DDEVICEDESCFLAGS_d3d_vb

CONST_D3DDEVINFOID

The **CONST_D3DDEVINFOID** enumeration defines flags used with the **Direct3DDevice7.GetInfo** method.

```
Enum CONST_D3DDEVINFOID
    D3DDEVINFOID_D3DTEXTUREMANAGER = 2
    D3DDEVINFOID_TEXTUREMANAGER = 1
    D3DDEVINFOID_TEXTUREING = 3
End Enum
```

D3DDEVINFOID_D3DTEXTUREMANAGER

The type in the *DevInfoType* parameter of the **GetInfo** method is a **D3DDEVINFO_TEXTUREMANAGER** type that contains information about texture management that is performed by Direct3D.

D3DDEVINFOID_TEXTUREMANAGER

The type in the *DevInfoType* parameter of the **GetInfo** method is a **D3DDEVINFO_TEXTUREMANAGER** type that contains information about texture management being performed by the driver. If the driver does not manage textures, information about texture management performed by Direct3D is retrieved.

D3DDEVINFOID_TEXTUREING

The type in the *DevInfoType* parameter of the **GetInfo** method is a **D3DDEVINFO_TEXTUREING** type that contains information about texturing activity of the application.

See Also

Direct3DDevice7.GetInfo

CONST_D3DDPFLAGS

The **CONST_D3DDPFLAGS** enumeration defines flags that are used to determine the behavior of the Direct3D rendering methods. These flags are used with the **Direct3DDevice7.DrawPrimitive**, **Direct3DDevice7.DrawIndexedPrimitive**, **Direct3DDevice7.DrawPrimitiveVB**, and **Direct3DDevice7.DrawIndexedPrimitiveVB**.

```
Enum CONST_D3DDPFLAGS
    D3DDP_DEFAULT = 0
    D3DDP_WAIT = 1
End Enum
```

D3DDP_DEFAULT

IDH_CONST_D3DDEVINFOID_d3d_vb

IDH_CONST_D3DDPFLAGS_d3d_vb

Perform normal rendering. (Return as soon as the polygons are sent to the card.)

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

CONST_D3DFILLMODE

The **CONST_D3DFILLMODE** enumeration contains constants describing the fill mode. These values are used by the **D3DRENDERSTATE_FILLMODE** render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

Enum **CONST_D3DFILLMODE**

D3DFILL_POINT = 1
D3DFILL_WIREFRAME = 2
D3DFILL_SOLID = 3

End Enum

D3DFILL_POINT

Fill points.

D3DFILL_WIREFRAME

Fill wireframes. This fill mode currently does not work for clipped primitives when you are using the **DrawPrimitive** methods.

D3DFILL_SOLID

Fill solids.

CONST_D3DFOGMODE

The **CONST_D3DFOGMODE** enumeration contains constants describing the fog mode. These values are used by the **D3DRENDERSTATE_FOGTABLEMODE** and **D3DRENDERSTATE_FOGVERTEXMODE** render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

Enum **CONST_D3DFOGMODE**

D3DFOG_NONE = 0
D3DFOG_EXP = 1
D3DFOG_EXP2 = 2
D3DFOG_LINEAR = 3

End Enum

IDH_CONST_D3DFILLMODE_d3d_vb

IDH_CONST_D3DFOGMODE_d3d_vb

D3DFOG_NONE

No fog effect.

D3DFOG_EXP

The fog effect intensifies exponentially, according to the following formula:

$$f = 1 / e^{d \times \text{density}}$$

D3DFOG_EXP2

The fog effect intensifies exponentially with the square of the distance, according to the following formula:

$$f = 1 / e^{(d \times \text{density})^2}$$

D3DFOG_LINEAR

The fog effect intensifies linearly between the start and end points, according to the following formula:

$$f = \frac{\text{end} - a}{\text{end} - \text{start}}$$

This is the only fog mode currently supported.

Remarks

In monochromatic (ramp) lighting mode, fog works properly only when the fog color is black. (If there is no lighting, any fog color will work, since in this case any fog color is effectively black.)

For more information about fog, see Fog.

Note

Fog can be considered a measure of visibility — the lower the fog value produced by one of the fog equations, the less visible an object is.

CONST_D3DFVFCAPSFLAGS

The **CONST_D3DFVFCAPSFLAGS** enumeration defines values present within or used with the **IFVFCaps** member of the **D3DDEVICEDESC7** type.

```
Enum CONST_D3DFVFCAPSFLAGS
    D3DFVFCAPS_DONOTSTRIPELEMENTS = 524288
    D3DFVFCAPS_TEXCOORDCOUNTMASK = 65535
End Enum
```

```
# IDH_CONST_D3DFVFCAPSFLAGS_d3d_vb
```

D3DFVFCAPS_DONOTSTRIPELEMENTS

Device prefers that vertex elements not be stripped. That is, if the vertex format contains elements that will not be 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 will provide better performance.

D3DFVFCAPS_TEXCOORDCOUNTMASK

Used to mask the low 16-bits of the **IFVFCaps** member of the **D3DDEVICEDESC7** type. 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 only blend using the specified number of texture coordinate sets.)

Remarks

The following Visual Basic code fragment masks the low 16-bits of the **IFVFCaps** member to determine the total number of texture coordinate sets that the device can simultaneously use for multiple texture blending:

```
Dim HALdesc As D3DDEVICEDESC7, HELdesc As D3DDEVICEDESC7
Dim iTexCoords As Integer

' The d3ddevice variable is assumed to be referece to
' a valid Direct3DDevice3 object.
Call d3ddevice.GetCaps(HALDesc, HELDesc)

' Mask the low 16-bits and set them to an integer variable.
iTexCoords = (HALDesc.IFVFCaps And D3DFVFCAPS_TEXCOORDCOUNTMASK)
```

CONST_D3DFVFFLAGS

The **CONST_D3DFVFFLAGS** enumeration defines flexible vertex format flags for use with the DrawPrimitive rendering methods. For details, see Flexible Vertex Format Flags.

CONST_D3DIMERR

The **CONST_D3DIMERR** enumeration defines error codes raised by the system, and are not otherwise useful. For descriptions of these error codes, see Error Codes.

```
# IDH_CONST_D3DFVFFLAGS_d3d_vb
# IDH_CONST_D3DIMERR_d3d_vb
```

CONST_D3DLIGHTCAPSFLAGS

The **CONST_D3DLIGHTCAPSFLAGS** enumeration defines lighting capability flags that are combined and present in the **ICaps** member of the **D3DLIGHTINGCAPS** type.

```
Enum CONST_D3DLIGHTCAPSFLAGS
    D3DLIGHTCAPS_DIRECTIONAL = 4
    D3DLIGHTCAPS_POINT = 1
    D3DLIGHTCAPS_SPOT = 2
End Enum
```

D3DLIGHTCAPS_DIRECTIONAL
Supports directional lights.

D3DLIGHTCAPS_POINT
Supports point lights.

D3DLIGHTCAPS_SPOT
Supports spotlights.

CONST_D3DLIGHTINGMODELFLAGS

The **CONST_D3DLIGHTINGMODELFLAGS** enumeration defines lighting model capability flags that are combined and present in the **ILightingModel** member of the **D3DLIGHTINGCAPS** type.

```
Enum CONST_D3DLIGHTINGMODELFLAGS
    D3DLIGHTINGMODEL_MONO = 2
    D3DLIGHTINGMODEL_RGB = 1
End Enum
```

D3DLIGHTINGMODEL_MONO
Monochromatic lighting model.

D3DLIGHTINGMODEL_RGB
RGB lighting model.

CONST_D3DLIGHTTYPE

The **CONST_D3DLIGHTTYPE** enumeration defines flags that identify light types. These flags are used in the **IType** member of the **D3DLIGHT7** type.

```
Enum CONST_D3DLIGHTTYPE
```

```
# IDH_CONST_D3DLIGHTCAPSFLAGS_d3d_vb
# IDH_CONST_D3DLIGHTINGMODELFLAGS_d3d_vb
# IDH_CONST_D3DLIGHTTYPE_d3d_vb
```

```

D3DLIGHT_DIRECTIONAL = 3
D3DLIGHT_POINT = 1
D3DLIGHT_SPOT = 2
End Enum

```

D3DLIGHT_DIRECTIONAL

Light is a directional source. This is equivalent to using a point light source at an infinite distance.

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 something like a point light except that the illumination is limited to a cone. This light type has a direction and several other parameters which determine the shape of the cone it produces. For information about these parameters, see the **D3DLIGHT7** type.

CONST_D3DMATERIALCOLORSOURCE

The **CONST_D3DMATERIALCOLORSOURCE** enumeration defines the location at which a color or color component should be accessed for lighting calculations.

```

Enum CONST_D3DMATERIALCOLORSOURCE
    D3DMCS_MATERIAL    = 0
    D3DMCS_COLOR1      = 1
    D3DMCS_COLOR2      = 2
End Enum

```

D3DMCS_MATERIAL

Use the color from the current material.

D3DMCS_COLOR1

Use the diffuse vertex color.

D3DMCS_COLOR2

Use the specular vertex color.

Remarks

The values defined by this enumeration are used with the **D3DRENDERSTATE_DIFFUSEMATERIALSOURCE**, **D3DRENDERSTATE_SPECULARMATERIALSOURCE**,

IDH_CONST_D3DMATERIALCOLORSOURCE_d3d_vb

D3DRENDERSTATE_AMBIENTMATERIALSOURCE, and
D3DRENDERSTATE_EMISSIVEMATERIALSOURCE render states.

CONST_D3DPRIMITIVETYPE

The **CONST_D3DPRIMITIVETYPE** enumeration lists the primitives supported by DrawPrimitive methods.

```
Enum CONST_D3DPRIMITIVETYPE
    D3DPT_POINTLIST    = 1
    D3DPT_LINELIST     = 2
    D3DPT_LINESTRIP    = 3
    D3DPT_TRIANGLELIST = 4
    D3DPT_TRIANGLESTRIP = 5
    D3DPT_TRIANGLEFAN  = 6
End Enum
```

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 will 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 will fail if the count is less than 2.

D3DPT_TRIANGLELIST

Renders the specified vertices as a sequence of isolated triangles. Each group of 3 vertices defines a separate triangle. Calls using this primitive type will fail if the count is less than 3, or if not evenly divisible by 3.

Back-face culling is affected by the current winding order render state.

D3DPT_TRIANGLESTRIP

Renders the vertices as a triangle strip. Calls using this primitive type will fail if the count is less than 3. The back-face removal flag is automatically flipped on even numbered triangles.

D3DPT_TRIANGLEFAN

Renders the vertices as a triangle fan. Calls using this primitive type will fail if the count is less than 3.

Remarks

Using triangle strips or fans is often more efficient than using triangle lists, as fewer vertices are duplicated. For a conceptual overview and information about defining triangle strips and fans, see Triangle Strips and Triangle Fans.

IDH_CONST_D3DPRIMITIVETYPE_d3d_vb

See Also

Direct3DDevice7.DrawIndexedPrimitive, **Direct3DDevice7.DrawPrimitive**,
Primitive Types

CONST_D3DPROCESSVERTICESFL LAGS

The **CONST_D3DPROCESSVERTICESFLAGS** enumeration defines flags used with the **Direct3DVertexBuffer7.ProcessVertices** method.

```
Enum CONST_D3DPROCESSVERTICESFLAGS
```

```
    D3DPV_DEFAULT = 0
```

```
    D3DPV_DONOTCOPYDATA = 1
```

```
End Enum
```

D3DPV_DEFAULT

Perform standard vertex processing using the current vertex operation.

D3DPV_DONOTCOPYDATA

Process the vertices into the destination buffer, but do not copy vertex data not affected by the current vertex operation into the destination buffer.

See Also

Direct3DVertexBuffer7.ProcessVertices, Processing Vertices

CONST_D3DRENDERSTATESINGL E

The **CONST_D3DRENDERSTATESINGLE** enumeration defines device render states that are set and retrieved through the **Direct3DDevice7.SetRenderStateSingle** and **Direct3DDevice7.GetRenderStateSingle** methods.

```
Type CONST_D3DRENDERSTATESINGLE
```

```
    D3DRENDERSTATE_FOGDENSITY = 38
```

```
    D3DRENDERSTATE_FOGEND = 37
```

```
    D3DRENDERSTATE_FOGSTART = 36
```

```
End Type
```

D3DRENDERSTATE_FOGDENSITY

Fog density for pixel or vertex fog to be used in the exponential fog modes (**D3DFOG_EXP** and **D3DFOG_EXP2**). Valid density values range from 0.0 to

```
# IDH_CONST_D3DPROCESSVERTICESFLAGS_d3d_vb
```

```
# IDH_CONST_D3DRENDERSTATESINGLE_d3d_vb
```


1.0, inclusive. The default value is 1.0. For more information, see Fog Parameters.

D3DRENDERSTATE_FOGEND and D3DRENDERSTATE_FOGSTART

Depth at which pixel or vertex fog effects begin and 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.

These render states enable you to exclude fog effects for positions close to the camera. When using z-based pixel fog, for example, you could set the starting depth to 0.3 to prevent fog effects for depths between 0.0 and 0.299, and the ending depth to 0.7 to prevent additional fog effects for depths between 0.701 and 1.0.

See Also

Direct3DDevice7.SetRenderStateSingle, **Direct3DDevice7.GetRenderStateSingle**

CONST_D3DRENDERSTATETYPE

The **CONST_D3DRENDERSTATETYPE** enumeration defines device render states that are set and retrieved through the **Direct3DDevice7.SetRenderState** and **Direct3DDevice7.GetRenderState** methods.

```
Enum CONST_D3DRENDERSTATETYPE
    D3DRENDERSTATE_ALPHABLENDENABLE    = 27
    D3DRENDERSTATE_ALPHAFUNC           = 25
    D3DRENDERSTATE_ALPHAREF             = 24
    D3DRENDERSTATE_ALPHATESTENABLE     = 15
    D3DRENDERSTATE_AMBIENT              = 139
    D3DRENDERSTATE_AMBIENTMATERIALSOURCE = 147
    D3DRENDERSTATE_ANTIALIAS            = 2
    D3DRENDERSTATE_CLIPPING             = 136
    D3DRENDERSTATE_CLIPPLANEENABLE     = 152
    D3DRENDERSTATE_COLORKEYBLENDENABLE = 144
    D3DRENDERSTATE_COLORKEYENABLE      = 41
    D3DRENDERSTATE_COLORVERTEX          = 141
    D3DRENDERSTATE_CULLMODE             = 22
    D3DRENDERSTATE_DESTBLEND            = 20
    D3DRENDERSTATE_DIFFUSEMATERIALSOURCE = 145
    D3DRENDERSTATE_DITHERENABLE        = 26
    D3DRENDERSTATE_EDGEANTIALIAS        = 40
    D3DRENDERSTATE_EMISSIVEMATERIALSOURCE = 148
```

IDH_CONST_D3DRENDERSTATETYPE_d3d_vb

```

D3DRENDERSTATE_EXTENTS          = 138
D3DRENDERSTATE_FILLMODE         = 8
D3DRENDERSTATE_FOGCOLOR        = 34
D3DRENDERSTATE_FOGENABLE       = 28
D3DRENDERSTATE_FOGTABLEMODE    = 35
D3DRENDERSTATE_FOGVERTEXMODE   = 140
D3DRENDERSTATE_LASTPIXEL       = 16
D3DRENDERSTATE_LIGHTING        = 137
D3DRENDERSTATE_LINEPATTERN     = 10
D3DRENDERSTATE_LOCALVIEWER     = 142
D3DRENDERSTATE_NORMALIZENORMALS = 143
D3DRENDERSTATE_RANGEFOGENABLE  = 48
D3DRENDERSTATE_SHADEMODE       = 9
D3DRENDERSTATE_SPECULARENABLE  = 29
D3DRENDERSTATE_SPECULARMATERIALSOURCE = 146
D3DRENDERSTATE_SRCBLEND        = 19
D3DRENDERSTATE_STENCILENABLE    = 52
D3DRENDERSTATE_STENCILFAIL     = 53
D3DRENDERSTATE_STENCILFUNC     = 56
D3DRENDERSTATE_STENCILMASK     = 58
D3DRENDERSTATE_STENCILPASS     = 55
D3DRENDERSTATE_STENCILREF      = 57
D3DRENDERSTATE_STENCILWRITEMASK = 59
D3DRENDERSTATE_STENCILZFAIL    = 54
D3DRENDERSTATE_STIPPLEDALPHA   = 33
D3DRENDERSTATE_TEXTUREFACTOR   = 60
D3DRENDERSTATE_TEXTUREPERSPECTIVE = 4
D3DRENDERSTATE_VERTEXBLEND     = 151
D3DRENDERSTATE_WRAP0          = 128
' Wrap render states 1 through 6 omitted here.
D3DRENDERSTATE_WRAP7          = 135
D3DRENDERSTATE_ZBIAS          = 47
D3DRENDERSTATE_ZENABLE        = 7
D3DRENDERSTATE_ZFUNC          = 23
D3DRENDERSTATE_ZVISIBLE       = 30
D3DRENDERSTATE_ZWRITEENABLE   = 14
End Type

```

D3DRENDERSTATE_ALPHABLENDENABLE

True to enable alpha-blended transparency. The default value is False. This member supersedes the legacy D3DRENDERSTATE_BLENDENABLE render state; see remarks for more information.

You can use the D3DRENDERSTATE_COLORKEYENABLE render state to toggle color keying. (Hardware rasterizers have always used the D3DRENDERSTATE_BLENDENABLE render state only for toggling alpha blending.)

The type of alpha blending is determined by the D3DRENDERSTATE_SRCBLEND and D3DRENDERSTATE_DESTBLEND render states. D3DRENDERSTATE_ALPHABLENDENABLE, with D3DRENDERSTATE_COLORKEYENABLE, allows fine blending control.

Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **D3DDEVICEDESC7** type to find out whether this render state is supported.

D3DRENDERSTATE_ALPHAFUNC

One of the constants 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.

D3DRENDERSTATE_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 0x00010000. The default value is 0.0.

D3DRENDERSTATE_ALPHATESTENABLE

True to enable alpha tests. The default value is False. This member enables applications to turn off the tests that otherwise would 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 D3DRENDERSTATE_ALPHAFUNC render state. When this mode is enabled, alpha blending occurs only if the test succeeds.

D3DRENDERSTATE_AMBIENT

Ambient light color. The default value is 0. This render state is analogous to the legacy D3DLIGHTSTATE_AMBIENT lighting state.

D3DRENDERSTATE_AMBIENTMATERIALSOURCE

The ambient color source for lighting calculations. Valid values are members of the **CONST_D3DMATERIALCOLORSOURCE** enumeration. The default value is D3DMCS_COLOR2.

D3DRENDERSTATE_ANTI_ALIAS

One of the members of the **CONST_D3DANTI_ALIASMODE** enumeration specifying the desired type of full-scene antialiasing. The default value is D3DANTI_ALIAS_NONE. For more information, see Full-scene Antialiasing and Antialiasing States.

You can only enable full-scene antialiasing on devices that expose the D3DPRASERCAPS_ANTI_ALIAS_SORTINDEPENDENT or D3DPRASERCAPS_ANTI_ALIAS_SORTDEPENDENT capabilities.

D3DRENDERSTATE_CLIPPING

Nonzero to enable primitive clipping by Direct3D, or zero to disable it. The default value is nonzero.

D3DRENDERSTATE_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 isn't included, the clipping plane is disabled. The default value is `D3DCPF_DISABLEALL`.

`D3DRENDERSTATE_COLORKEYBLENDENABLE`

Nonzero to enable alpha-blended colorkeying, or zero to disable it. Alpha-blended colorkey mode sets the alpha and color of matching texture samples to zero. The application is required to set the alpha test and alpha blend modes to achieve the desired discard and/or alpha-blended result.

`D3DRENDERSTATE_COLORKEYENABLE`

True to enable color-keyed transparency. The default value is False. You can use this render state with `D3DRENDERSTATE_ALPHABLENDENABLE` to implement fine blending control.

Applications should check the `D3DDEVCAPS_DRAWPRIMTLVERTEX` flag in the **`D3DDEVICEDESC7`** type to find out whether this render state is supported.

When color-keyed transparency is enabled, only texture surfaces that were created with the `DDSD_CKSRCLT` flag will be affected. Surfaces that were created without the `DDSD_CKSRCLT` flag will exhibit color-keyed transparency effects.

`D3DRENDERSTATE_COLORVERTEX`

Nonzero to enable per-vertex color, or zero to disable it. The default value is nonzero. Enabling per-vertex color allows the system to include the color defined for individual vertices in its lighting calculations.

See also: `D3DRENDERSTATE_DIFFUSEMATERIALSOURCE`, `D3DRENDERSTATE_SPECULARMATERIALSOURCE`, `D3DRENDERSTATE_AMBIENTMATERIALSOURCE`, and `D3DRENDERSTATE_EMISSIVEMATERIALSOURCE`.

`D3DRENDERSTATE_CULLMODE`

Specifies how back-facing triangles are to be culled, if at all. This can be set to one of the constants of the **`CONST_D3DCULL`** enumeration. The default value is `D3DCULL_CCW`.

`D3DRENDERSTATE_DESTBLEND`

One of the constants of the **`CONST_D3DBLEND`** enumeration. The default value is `D3DBLEND_ZERO`.

`D3DRENDERSTATE_DIFFUSEMATERIALSOURCE`

The 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 `D3DRENDERSTATE_COLORVERTEX` render state is set to nonzero.

`D3DRENDERSTATE_DITHERENABLE`

True to enable dithering. The default value is False.

`D3DRENDERSTATE_EDGEANTIALIAS`

True to antialias lines forming the convex outline of objects. The default value is False. For more information, see *Edge Antialiasing and Antialiasing States*. When set to True, applications should only render 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 simply 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 only enable edge antialiasing on devices that expose the `D3DPRASTERCAPS_ANTIALIASEDGEDGES` capability.

`D3DRENDERSTATE_EMISSIVEMATERIALSOURCE`

The emissive color source for lighting calculations. Valid values are members of the `CONST_D3DMATERIALCOLORSOURCE` enumeration. The default value is `D3DMCS_MATERIAL`.

`D3DRENDERSTATE_EXTENTS`

Nonzero to cause the system to update the screen extents for each rendering call, or zero to disable screen extent updates. The default value is zero.

`D3DRENDERSTATE_FILLMODE`

One or more constants of the `CONST_D3DFILLMODE` enumeration. The default value is `D3DFILL_SOLID`.

`D3DRENDERSTATE_FOGCOLOR`

Value of type Long indicating RGB color, as returned by the **DirectX7.CreateColorRGB** method. The default value is 0. For more information, see Fog Color.

`D3DRENDERSTATE_FOGENABLE`

True to enable fog blending. The default value is False. For more information, see Fog Blending and Fog.

These render states enable you to exclude fog effects for positions close to the camera. For example, you could set the starting depth to 0.3 to prevent fog effects for depths between 0.0 and 0.299, and the ending depth to 0.7 to prevent additional fog effects for depths between 0.701 and 1.0.

`D3DRENDERSTATE_FOGTABLEMODE`

The fog formula to be used for pixel fog. Set to one of the constants of the `CONST_D3DFOGMODE` enumeration. The default value is `D3DFOG_NONE`. For more information, see Pixel Fog.

`D3DRENDERSTATE_FOGVERTEXMODE`

The fog formula to be used for vertex fog. Set to one of the members of the `CONST_D3DFOGMODE` enumeration. The default value is `D3DFOG_NONE`. For more information, see Vertex Fog.

`D3DRENDERSTATE_LASTPIXEL`

False to enable drawing the last pixel in a line or triangle. The default value is True.

`D3DRENDERSTATE_LIGHTING`

Nonzero to enable Direct3D lighting, or zero disable it. The default value is nonzero. Only vertices that include a vertex normal will be properly lit; vertices that do not contain a normal will employ a dot-product of 0 in all lighting calculations.

D3DRENDERSTATE_LINEPATTERN

Line pattern. The **D3DLINEPATTERN** type defines valid values. The default values for the members of **D3DLINEPATTERN** are 0 for **repeatPattern** and 0 for **linePattern**.

D3DRENDERSTATE_LOCALVIEWER

Nonzero to enable camera-relative specular highlights, or zero to use orthogonal specular highlights. The default value is nonzero to use camera-relative specular highlights. Applications that use orthogonal projection should specify zero.

D3DRENDERSTATE_NORMALIZENORMALS

Nonzero to enable automatic normalization of vertex normals, or zero to disable it. The default value is zero. Enabling this feature causes the system to normalize the vertex normals for vertices after transforming them to camera space, which can be computationally intensive.

D3DRENDERSTATE_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 eye is rotated — in this case, the depth changes while the range remains constant.

This render state works only with **D3DVERTEX** vertices. When you specify **D3DLVERTEX** or **D3DTLVERTEX** vertices, the F (fog) component of the RGBF fog value should already be corrected for range.

Since no hardware currently supports per-pixel range-based fog, range correction offered only for vertex fog. For more information, see Range-based Fog and Vertex Fog

D3DRENDERSTATE_SHADEMODE

One or more constants of the **D3DSHADEMODE** enumeration. The default value is **D3DSHADE_GOURAUD**.

D3DRENDERSTATE_SPECULARENABLE

True to enable specular highlights. The default value is True.

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.

D3DRENDERSTATE_SPECULARMATERIALSOURCE

The specular color source for lighting calculations. Valid values are members of the **CONST_D3DMATERIALCOLORSOURCE** enumeration. The default value is **D3DMCS_COLOR2**.

D3DRENDERSTATE_SRCBLEND

One of the constants of the **CONST_D3DBLEND** enumeration. The default value is D3DBLEND_ONE.

D3DRENDERSTATE_STENCILENABLE

True to enable stenciling, or False to disable stenciling. The default value is False. For more information, see Stencil Buffers.

D3DRENDERSTATE_STENCILFAIL

Stencil operation to perform if the stencil test fails. This can be one of the constants of the **CONST_D3DSTENCILOP** enumeration. The default value is D3DSTENCILOP_KEEP. For more information, see Stencil Buffers.

D3DRENDERSTATE_STENCILFUNC

Comparison function for the stencil test. This can be one of the constants 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 only applies to the bits in the reference value and stencil buffer entry that are set in the stencil mask (set by the D3DRENDERSTATE_STENCILMASK render state). If the comparison is true, the stencil test passes.

D3DRENDERSTATE_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.

D3DRENDERSTATE_STENCILPASS

Stencil operation to perform if both the stencil and depth (z) tests pass. This can be one of the constants of the **CONST_D3DSTENCILOP** enumeration. The default value is D3DSTENCILOP_KEEP. For more information, see Stencil Buffers.

D3DRENDERSTATE_STENCILREF

Integer reference value for the stencil test. The default value is 0.

D3DRENDERSTATE_STENCILWRITEMASK

Write mask applied to values written into the stencil buffer. The default mask is 0xFFFFFFFF.

D3DRENDERSTATE_STENCILZFAIL

Stencil operation to perform if the stencil test passes and depth test (z-test) fails. This can be one of the constants of the **CONST_D3DSTENCILOP** enumeration. The default value is D3DSTENCILOP_KEEP. For more information, see Stencil Buffers.

D3DRENDERSTATE_STIPPLEDALPHA

True to enable stippled alpha. The default value is False.

Current software rasterizers ignore this render state. You can use the D3DPSHADECAPS_ALPHAFLATSTIPPLED flag in the **D3DPRIMCAPS** type to discover whether the current hardware supports this render state.

D3DRENDERSTATE_TEXTUREFACTOR

Color used for multiple texture blending with the **D3DTA_TFACTOR** texture-blending argument or **D3DTOP_BLENDFACTORALPHA** texture-blending operation.

D3DRENDERSTATE_TEXTUREPERSPECTIVE

True to enable for perspective correct texture mapping. (See perspective correction.) The default value is True. For more information, see Texture Perspective State.

D3DRENDERSTATE_VERTEXBLEND

Number of matrices to be used to perform geometry blending, if any. Valid values are members of the **CONST_D3DVERTEXBLEND_FLAGS** enumeration. The default value is **D3DVBLEND_DISABLE**. For more information see Geometry Blending.

D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7

Texture wrapping behavior for multiple sets of texture coordinates. Valid values for these render states can be any combination of the **D3DWRAPCOORD_0**, **D3DWRAPCOORD_1**, **D3DWRAPCOORD_2**, and **D3DWRAPCOORD_3** flags from the **CONST_D3D** 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). For more information, see Texture Wrapping.

D3DRENDERSTATE_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. For more information, see Using Depth Buffers.

D3DRENDERSTATE_ZENABLE

The depth buffering state, as one of the constants 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 buffer is attached to the render-target surface, and **D3DZB_FALSE** otherwise.

D3DRENDERSTATE_ZFUNC

One of the constants 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 **D3DRENDERSTATE_ZWRITEENABLE** is True.

Software rasterizers and many hardware accelerators work faster if the depth test fails, since there is no need to filter and modulate the texture if the pixel is not going to be rendered.

D3DRENDERSTATE_ZVISIBLE

Not used.

D3DRENDERSTATE_ZWRITEENABLE

True to enable writes 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 this state is False, depth comparisons are still made according to the render state D3DRENDERSTATE_ZFUNC (assuming depth buffering is taking place), but depth values are not written to the buffer.

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 D3DRENDERSTATE_WRAP n state values). Add the D3DRENDERSTATE_WRAPBIAS value to the zero-based index of a texture coordinate set to calculate the D3DRENDERSTATE_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 d3dDevice.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_D3DSHADEMODE

The **CONST_D3DSHADEMODE** enumeration describes the supported shade mode for the D3DRENDERSTATE_SHADEMODE render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

```
Enum CONST_D3DSHADEMODE
    D3DSHADE_FLAT      = 1
    D3DSHADE_GOURAUD   = 2
    D3DSHADE_PHONG     = 3
End Enum
```

```
# IDH_CONST_D3DSHADEMODE_d3d_vb
```

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 aren't interpolated.

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.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DSTATEBLOCKTYPE

The **CONST_D3DSTATEBLOCK** enumerated type defines logical groups of device states, for use with the **Direct3DDevice7.CreateStateBlock** method.

Type **D3DSTATEBLOCKTYPE**

D3DSBT_ALL = 1

D3DSBT_PIXELSTATE = 2

D3DSBT_VERTEXSTATE = 3

End Type

D3DSBT_ALL

Capture all device states.

D3DSBT_PIXELSTATE

Capture only pixel-related device states.

D3DSBT_VERTEXSTATE

Capture only vertex-related device states.

Remarks

The **D3DSBT_PIXELSTATE** and **D3DSBT_VERTEXSTATE** values identify different logical groups of device states, though some states are common to both groups. For information about the states defined by each group, see **Creating Predefined State Blocks**.

See Also

Direct3DDevice7.CreateStateBlock

IDH_CONST_D3DSTATEBLOCKTYPE_d3d_vb

CONST_D3DSTENCILCAPSFLAGS

The **CONST_D3DSTENCILCAPSFLAGS** enumeration defines stencil buffer capability flags that are combined and present in the **IStencilCaps** member of the **D3DDEVICEDESC7** type.

Enum **CONST_D3DSTENCILCAPSFLAGS**

D3DSTENCILCAPS_DECR = 128
D3DSTENCILCAPS_DECRSAT = 16
D3DSTENCILCAPS_INCR = 64
D3DSTENCILCAPS_INCRSAT = 8
D3DSTENCILCAPS_INVERT = 32
D3DSTENCILCAPS_KEEP = 1
D3DSTENCILCAPS_REPLACE = 4
D3DSTENCILCAPS_ZERO = 2

End Enum

D3DSTENCILCAPS_DECR

The **D3DSTENCILOP_DECR** stencil buffer operation is supported.

D3DSTENCILCAPS_DECRSAT

The **D3DSTENCILOP_DECRSAT** stencil buffer operation is supported.

D3DSTENCILCAPS_INCR

The **D3DSTENCILOP_INCR** stencil buffer operation is supported.

D3DSTENCILCAPS_INCRSAT

The **D3DSTENCILOP_INCRSAT** stencil buffer operation is supported.

D3DSTENCILCAPS_INVERT

The **D3DSTENCILOP_INVERT** stencil buffer operation is supported.

D3DSTENCILCAPS_KEEP

The **D3DSTENCILOP_KEEP** stencil buffer operation is supported.

D3DSTENCILCAPS_REPLACE

The **D3DSTENCILOP_REPLACE** stencil buffer operation is supported.

D3DSTENCILCAPS_ZERO

The **D3DSTENCILOP_ZERO** stencil buffer operation is supported.

CONST_D3DSTENCILOP

The **CONST_D3DSTENCILOP** enumeration describes the stencil operations for the **D3DRENDERSTATE_STENCILFAIL**, **D3DRENDERSTATE_STENCILZFAIL**, **D3DRENDERSTATE_STENCILPASS** render states.

Enum **CONST_D3DSTENCILOP**

D3DSTENCILOP_DECR = 8

IDH_CONST_D3DSTENCILCAPSFLAGS_d3d_vb

IDH_CONST_D3DSTENCILOP_d3d_vb

```

D3DSTENCILOP_DECRSAT = 5
D3DSTENCILOP_INCR = 7
D3DSTENCILOP_INCRSAT = 4
D3DSTENCILOP_INVERT = 6
D3DSTENCILOP_KEEP = 1
D3DSTENCILOP_REPLACE = 3
D3DSTENCILOP_ZERO = 2
End Enum

```

D3DSTENCILOP_DECR

Decrement the stencil-buffer entry, wrapping to the maximum value if the new value is less than zero.

D3DSTENCILOP_DECRSAT

Decrement the stencil-buffer entry, clamping to zero.

D3DSTENCILOP_INCRSAT

Increment the stencil-buffer entry, clamping to the maximum value. See remarks for information on the maximum stencil-buffer values.

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_KEEP

Do not update the entry in the stencil buffer. This is the default value.

D3DSTENCILOP_REPLACE

Replace the stencil-buffer entry with reference value.

D3DSTENCILOP_ZERO

Set the stencil-buffer entry to zero.

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.

See Also

CONST_D3DRENDERSTATETYPE, Stencil Buffers

CONST_D3DTAFLAGS

The **CONST_D3DTAFLAGS** enumeration defines texture argument flags used to define texture blending stages. For details, see Texture Argument Flags.

IDH_CONST_D3DTAFLAGS_d3d_vb

CONST_D3DTEXCOORDINDEXFLA GS

The **CONST_D3DTEXCOORDINDEXFLAGS** enumeration defines values used in combination with the index values set with the **D3DTSS_TEXCOORDINDEX** texture-stage state.

Type **CONST_D3DTEXCOORDINDEXFLAGS**

D3DTSS_TCI_CAMERASPACENORMAL = 65536

D3DTSS_TCI_CAMERASPACEPOSITION = 131072

D3DTSS_TCI_CAMERASPACE REFLECTIONVECTOR = 196608

D3DTSS_TCI_PASSTHRU = 0

End Type

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_CAMERASPACE REFLECTIONVECTOR

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_TCI_PASSTHRU

Use the specified texture coordinates contained within the vertex format.

Remarks

The values from this enumeration (except for **D3DTSS_TCI_PASSTHRU**, which resolves to zero) change how the system uses the index value being set in **D3DTSS_TEXCOORDINDEX**. Normally, the value set in **D3DTSS_TEXCOORDINDEX** controls the texture coordinate set used to address the texture. However, when you include the **D3DTSS_TCI_CAMERASPACENORMAL**, **D3DTSS_TCI_CAMERASPACEPOSITION**, or **D3DTSS_TCI_CAMERASPACE REFLECTIONVECTOR** values (as a logical **Or** operation with the index being set), the system uses the index strictly to determine the texture wrapping mode.

Values within this enumeration are useful to applications that benefit from automatic generation of texture coordinates for environment mapping.

IDH_CONST_D3DTEXCOORDINDEXFLAGS_d3d_vb

See Also

Configuring Automatically Generated Texture Coordinates

CONST_D3DTEXOPCAPSFLAGS

The **CONST_D3DTEXOPCAPSFLAGS** enumeration defines texture-blending operation capabilities that are combined and present in the **ITextureOpCaps** member of the **D3DDEVICEDESC7** type.

```
Enum CONST_D3DTEXOPCAPSFLAGS
    D3DTEXOPCAPS_ADD = 64
    D3DTEXOPCAPS_ADDSIGNED = 128
    D3DTEXOPCAPS_ADDSIGNED2X = 256
    D3DTEXOPCAPS_ADDSMOOTH = 1024
    D3DTEXOPCAPS_BLENDCURRENTALPHA = 32768
    D3DTEXOPCAPS_BLENDDIFFUSEALPHA = 2048
    D3DTEXOPCAPS_BLENDFACTORALPHA = 8192
    D3DTEXOPCAPS_BLENDTEXTUREALPHA = 4096
    D3DTEXOPCAPS_BLENDTEXTUREALPHAPM = 16384
    D3DTEXOPCAPS_BUMPENVMAP = 2097152
    D3DTEXOPCAPS_BUMPENVMAPLUMINANCE = 4194304
    D3DTEXOPCAPS_DISABLE = 1
    D3DTEXOPCAPS_DOTPRODUCT3 = 8388608
    D3DTEXOPCAPS_MODULATE = 8
    D3DTEXOPCAPS_MODULATE2X = 16
    D3DTEXOPCAPS_MODULATE4X = 32
    D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR = 131072
    D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA = 262144
    D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR = 524288
    D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA = 1048576
    D3DTEXOPCAPS_PREMODULATE = 65536
    D3DTEXOPCAPS_SELECTARG1 = 2
    D3DTEXOPCAPS_SELECTARG2 = 4
    D3DTEXOPCAPS_SUBTRACT = 512
End Enum
```

D3DTEXOPCAPS_ADD

The **D3DTEXTOP_ADD** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSIGNED

The **D3DTEXTOP_ADDSIGNED** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSIGNED2X

The **D3DTEXTOP_ADDSIGNED2X** texture blending operation is supported by this device.

IDH_CONST_D3DTEXOPCAPSFLAGS_d3d_vb

D3DTEXOPCAPS_ADDSMOOTH

The **D3DTOP_ADDSMOOTH** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDCURRENTALPHA

The **D3DTOP_BLENDCURRENTALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDDIFFUSEALPHA

The **D3DTOP_BLENDDIFFUSEALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDFACTORALPHA

The **D3DTOP_BLENDFACTORALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDTEXTUREALPHA

The **D3DTOP_BLENDTEXTUREALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDTEXTUREALPHAPM

The **D3DTOP_BLENDTEXTUREALPHAPM** texture blending operation is supported by this device.

D3DTEXOPCAPS_BUMPENVMAP

The **D3DTOP_BUMPENVMAP** texture blending operation is supported by this device.

D3DTEXOPCAPS_BUMPENVMAPLUMINANCE

The **D3DTOP_BUMPENVMAPLUMINANCE** texture blending operation is supported by this device.

D3DTEXOPCAPS_DISABLE

The **D3DTOP_DISABLE** texture blending operation is supported by this device.

D3DTEXOPCAPS_DOTPRODUCT3

The **D3DTOP_DOTPRODUCT3** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE

The **D3DTOP_MODULATE** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE2X

The **D3DTOP_MODULATE2X** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE4X

The **D3DTOP_MODULATE4X** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR

The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA

The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR

The **D3DTOP_MODULATEINVALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA

The **D3DTOP_MODULATEINVCOLOR_ADDALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_PREMODULATE

The **D3DTOP_PREMODULATE** texture blending operation is supported by this device.

D3DTEXOPCAPS_SELECTARG1

The **D3DTOP_SELECTARG1** texture blending operation is supported by this device.

D3DTEXOPCAPS_SELECTARG2

The **D3DTOP_SELECTARG2** texture blending operation is supported by this device.

D3DTEXOPCAPS_SUBTRACT

The **D3DTOP_SUBTRACT** texture blending operation is supported by this device.

CONST_D3DTEXTUREADDRESS

The **CONST_D3DTEXTUREADDRESS** enumeration describes the supported texture addressing modes when setting them with the **D3DTSS_ADDRESS**, **D3DTSS_ADDRESSU**, and **D3DTSS_ADDRESSV** texture stage states.

Enum **CONST_D3DTEXTUREADDRESS**

D3DTADDRESS_WRAP = 1
D3DTADDRESS_MIRROR = 2
D3DTADDRESS_CLAMP = 3
D3DTADDRESS_BORDER = 4

End Enum

D3DTADDRESS_WRAP

Tile the texture at every integer junction. For example, for u values between 0 and 3, the texture will be 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

IDH_CONST_D3DTEXTUREADDRESS_d3d_vb

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 drawn with the border color, set with the D3DTSS_BORDERCOLOR texture stage state in the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

See Also

CONST_D3DTEXTURESTAGESTATETYPE

CONST_D3DTEXTUREMAGFILTER

The **CONST_D3DTEXTUREMAGFILTER** enumeration defines texture magnification filtering modes for a texture stage.

```
Enum CONST_D3DTEXTUREMAGFILTER
```

```
    D3DTFG_POINT      = 1
```

```
    D3DTFG_LINEAR     = 2
```

```
    D3DTFG_FLATCUBIC  = 3
```

```
    D3DTFG_GAUSSIANCUBIC= 4
```

```
    D3DTFG_ANISOTROPIC = 5
```

```
End Enum
```

D3DTFG_POINT

Point filtering. The texel with coordinates nearest to the desired pixel value is used.

D3DTFG_LINEAR

Bilinear interpolation filtering. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

D3DTFG_FLATCUBIC

Not currently supported; do not use.

D3DTFG_GAUSSIANCUBIC

Not currently supported; do not use.

D3DTFG_ANISOTROPIC

Anisotropic texture filtering. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

Remarks

You set a texture stage's magnification filter by calling the **Direct3DDevice7.SetTextureStageState** method with the D3DTSS_MAGFILTER value as the second parameter, and one of constants of this enumeration as the third parameter.

IDH_CONST_D3DTEXTUREMAGFILTER_d3d_vb

See Also

CONST_D3DTEXTUREMINFILTER, **CONST_D3DTEXTUREMIPFILTER**,
Texture Filtering

CONST_D3DTEXTUREMINFILTER

The **CONST_D3DTEXTUREMINFILTER** enumeration defines texture minification filtering modes for a texture stage.

Enum **CONST_D3DTEXTUREMINFILTER**

D3DTFN_POINT = 1
D3DTFN_LINEAR = 2
D3DTFN_ANISOTROPIC = 3

End Enum

D3DTFN_POINT

Point filtering. The texel with coordinates nearest to the desired pixel value is used.

D3DTFN_LINEAR

Bilinear interpolation filtering. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

D3DTFN_ANISOTROPIC

Anisotropic texture filtering. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

Remarks

You set a texture stage's magnification filter by calling the **Direct3DDevice7.SetTextureStageState** method with the **D3DTSS_MINFILTER** value as the second parameter, and one of constants of this enumeration as the third parameter.

See Also

CONST_D3DTEXTUREMAGFILTER, **CONST_D3DTEXTUREMIPFILTER**,
Texture Filtering

CONST_D3DTEXTUREMIPFILTER

The **CONST_D3DTEXTUREMIPFILTER** enumeration defines texture mipmapping filtering modes for a texture stage.

Enum **CONST_D3DTEXTUREMIPFILTER**

IDH_CONST_D3DTEXTUREMINFILTER_d3d_vb

IDH_CONST_D3DTEXTUREMIPFILTER_d3d_vb

```

D3DTFP_NONE      = 1
D3DTFP_POINT     = 2
D3DTFP_LINEAR    = 3
End Enum

```

D3DTFP_NONE

Mipmapping disabled. The rasterizer should use the magnification filter instead.

D3DTFP_POINT

Nearest point mipmap filtering. The rasterizer uses the color from the texel of the nearest mipmap texture.

D3DTFP_LINEAR

Trilinear mipmap interpolation. The rasterizer linearly interpolates pixel color using the texels of the two nearest mipmap textures.

Remarks

You set a texture stage's magnification filter by calling the **Direct3DDevice7.SetTextureStageState** method with the D3DTSS_MIPFILTER value as the second parameter, and one of constants of this enumeration as the third parameter.

See Also

CONST_D3DTEXTUREMAGFILTER, **CONST_D3DTEXTUREMIPFILTER**,
Texture Filtering

CONST_D3DTEXTUREOP

The **CONST_D3DTEXTUREOP** enumeration defines per-stage texture blending operations. The constants of this type are used when setting color or alpha operations by using the D3DTSS_COLOROP or D3DTSS_ALPHAOP values with the **Direct3DDevice7.SetTextureStageState** method.

```

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

```

IDH_CONST_D3DTEXTUREOP_d3d_vb

```

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
End Enum

```

Control constants

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.

$$S_{\text{RGBA}} = \text{Arg1}$$

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.

$$S_{\text{RGBA}} = \text{Arg2}$$

Modulation constants

D3DTOP_MODULATE

Multiply the components of the arguments together.

$$S_{\text{RGBA}} = \text{Arg1} \times \text{Arg2}$$

D3DTOP_MODULATE2X

Multiply the components of the arguments and shift the products to the left one bit (effectively multiplying them by two) for brightening.

$$S_{\text{RGBA}} = (\text{Arg1} \times \text{Arg2}) \ll 1$$

D3DTOP_MODULATE4X

Multiply the components of the arguments and shift the products to the left two bits (effectively multiplying them by four) for brightening.

$$S_{\text{RGBA}} = (\text{Arg1} \times \text{Arg2}) \ll 2$$

Addition and Subtraction constants

D3DTOP_ADD

Add the components of the arguments.

$$S_{\text{RGBA}} = \text{Arg1} + \text{Arg2}$$

D3DTOP_ADDSIGNED

Add the components of the arguments with a -0.5 bias, making the effective range of values from -0.5 to 0.5.

$$S_{\text{RGBA}} = \text{Arg1} + \text{Arg2} - 0.5$$

D3DTOP_ADDSIGNED2X

Add the components of the arguments with a -0.5 bias, and shift the products to the left one bit.

$$S_{\text{RGBA}} = (\text{Arg1} + \text{Arg2} - 0.5) \ll 1$$

D3DTOP_SUBTRACT

Subtract the components of the second argument from those of the first argument.

$$S_{\text{RGBA}} = \text{Arg1} - \text{Arg2}$$

D3DTOP_ADDSMOOTH

Add the first and second arguments, then subtract their product from the sum.

$$\begin{aligned} S_{\text{RGBA}} &= \text{Arg1} + \text{Arg2} - \text{Arg1} \times \text{Arg2} \\ &= \text{Arg1} + \text{Arg2} (1 - \text{Arg1}) \end{aligned}$$

Linear alpha blending constants

D3DTOP_BLENDDIFFUSEALPHA

D3DTOP_BLENDTEXTUREALPHA

D3DTOP_BLENDFACTORALPHA

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 D3DRENDERSTATE_TEXTUREFACTOR render state, or the alpha taken from the previous texture stage (D3DTOP_BLENDCURRENTALPHA).

$$S_{\text{RGBA}} = \text{Arg } 1 \times (\text{Alpha}) + \text{Arg } 2 \times (1 - \text{Alpha})$$

D3DTOP_BLENDTEXTUREALPHAPM

Linearly blend a texture stage that uses premultiplied alpha.

$$S_{\text{RGBA}} = \text{Arg } 1 + \text{Arg } 2 \times (1 - \text{Alpha})$$

Specular mapping constants**D3DTOP_PREMODULATE**

Modulate this texture stage with the next texture stage.

D3DTOP_MODULATEALPHA_ADDCOLOR

Modulate the second argument's color using the first argument's alpha, then add the result to argument one. This operation is supported only for color operations (D3DTSS_COLOROP).

$$S_{\text{RGBA}} = \text{Arg } 1_{\text{RGB}} + \text{Arg } 1_{\text{A}} \times \text{Arg } 2_{\text{RGB}}$$

D3DTOP_MODULATECOLOR_ADDALPHA

Modulate the arguments, then add the first argument's alpha. This operation is supported only for color operations (D3DTSS_COLOROP).

$$S_{\text{RGBA}} = \text{Arg } 1_{\text{RGB}} \times \text{Arg } 2_{\text{RGB}} + \text{Arg } 1_{\text{A}}$$

D3DTOP_MODULATEINVALPHA_ADDCOLOR

Similar to D3DTOP_MODULATEALPHA_ADDCOLOR, but use the inverse of the first argument's alpha. This operation is supported only for color operations (D3DTSS_COLOROP).

$$S_{\text{RGBA}} = (1 - \text{Arg } 1_{\text{A}}) \times \text{Arg } 2_{\text{RGB}} + \text{Arg } 1_{\text{RGB}}$$

D3DTOP_MODULATEINVCOLOR_ADDALPHA

Similar to D3DTOP_MODULATECOLOR_ADDALPHA, but use the inverse of the first argument's color. This operation is supported only for color operations (D3DTSS_COLOROP).

$$S_{\text{RGBA}} = (1 - \text{Arg } 1_{\text{RGB}}) \times \text{Arg } 2_{\text{RGB}} + \text{Arg } 1_{\text{A}}$$

Bump mapping constants**D3DTOP_BUMPENVMAP**

Perform per-pixel bump-mapping using the environment map in the next texture stage (without luminance).

D3DTOP_BUMPENVMAPLUMINANCE

Perform per-pixel bump-mapping using the environment map in the next texture stage (with luminance).

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 only for color operations (D3DTSS_COLOROP).

$$S_{\text{RGBA}} = (\text{Arg1}_{\text{R}} \times \text{Arg2}_{\text{R}} + \text{Arg1}_{\text{G}} \times \text{Arg2}_{\text{G}} + \text{Arg1}_{\text{B}} \times \text{Arg2}_{\text{B}})$$

Remarks

In the preceding 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 one would be shown as Arg1_{A} .

See Also

Direct3DDevice7.GetTextureStageState, **Direct3DDevice7.SetTextureStageState**, **CONST_D3DTEXTURESTAGESTATETYPE**

CONST_D3DTEXTURESTAGESINGLE

The **CONST_D3DTEXTURESTAGESINGLE** enumeration defines texture stage states that accepts values of type **Single**. Constants of this enumeration are used with the **Direct3DDevice7.GetTextureStageStateSingle** and **Direct3DDevice7.SetTextureStageStateSingle** methods to retrieve and set texture state values.

Enum **CONST_D3DTEXTURESTAGESINGLE**

D3DTSS_BUMPENVLOFFSET = 23

D3DTSS_BUMPENVLSCALE = 22

D3DTSS_BUMPENVMAT00 = 7

D3DTSS_BUMPENVMAT01 = 8

D3DTSS_BUMPENVMAT10 = 9

D3DTSS_BUMPENVMAT11 = 10

D3DTSS_MIPMAPLODBIAS = 19

End Enum

D3DTSS_BUMPENVMAT00

The texture stage state is a value for the [0][0] coefficient in a bump mapping matrix. The default value is 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.

D3DTSS_BUMPENVMAT10

IDH_CONST_D3DTEXTURESTAGESINGLE_d3d_vb

The texture stage state is a value for the [1][0] coefficient in a bump mapping matrix. The default value is 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.

D3DTSS_BUMPENVLSCALE

Scale for bump map luminance. The default value is 0.

D3DTSS_BUMPENVLOFFSET

Offset for bump map luminance. The default value is 0.

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.

Remarks

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

Direct3DDevice7.GetTextureStageStateSingle,
Direct3DDevice7.SetTextureStageStateSingle

CONST_D3DTEXTURESTAGESTATETYPE

The **CONST_D3DTEXTURESTAGESTATETYPE** enumeration defines texture stage states. Constants of this enumeration are used with the **Direct3DDevice7.GetTextureStageState** and **Direct3DDevice7.SetTextureStageState** methods to retrieve and set texture state values.

Enum **CONST_D3DTEXTURESTAGESTATETYPE**

```
D3DTSS_ADDRESS      = 12
D3DTSS_ADDRESSU     = 13
D3DTSS_ADDRESSV     = 14
D3DTSS_ALPHAARG1    = 5
D3DTSS_ALPHAARG2    = 6
D3DTSS_ALPHAOP      = 4
D3DTSS_BORDERCOLOR  = 15
```

IDH_CONST_D3DTEXTURESTAGESTATETYPE_d3d_vb

```

D3DTSS_COLORARG1    = 2
D3DTSS_COLORARG2    = 3
D3DTSS_COLOROP      = 1
D3DTSS_MAGFILTER     = 16
D3DTSS_MAXANISOTROPY = 21
D3DTSS_MAXMIPLEVEL   = 20
D3DTSS_MINFILTER     = 17
D3DTSS_MIPFILTER     = 18
D3DTSS_TEXCOORDINDEX = 11
D3DTSS_TEXTURETRANSFORMFLAGS = 24
End Enum

```

D3DTSS_ADDRESS

Member of the **CONST_D3DTEXTUREADDRESS** enumeration. Selects the texture addressing method for both the u and v coordinates. The default is D3DADDRESS_WRAP.

D3DTSS_ADDRESSU

Member of the **CONST_D3DTEXTUREADDRESS** enumeration. Selects the texture addressing method for the u coordinate. The default is D3DADDRESS_WRAP.

D3DTSS_ADDRESSV

Member of the **CONST_D3DTEXTUREADDRESS** enumeration. Selects the texture addressing method for the v coordinate. The default value is D3DADDRESS_WRAP.

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.

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.

D3DTSS_ALPHAOP

The texture stage state is texture alpha blending operation identified by one of the constants 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_BORDERCOLOR

Value that describes the color to be used for rasterizing texture coordinates outside the [0.0,1.0] range. The default color is 0x00000000.

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.

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.

D3DTSS_COLOROP

The texture stage state is a texture color blending operation identified by one of the constants of the **CONST_D3DTEXTUREOP** enumeration. The default value for the first texture stage (stage 0) is D3DTOP_MODULATE, and for all other stages the default is D3DTOP_DISABLE.

D3DTSS_MAGFILTER

Member of the **CONST_D3DTEXTUREMAGFILTER** enumeration that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFG_POINT.

D3DTSS_MAXANISOTROPY

Maximum level of anisotropy. The default value is 1.

D3DTSS_MAXMIPLEVEL

Maximum level-of-detail mipmap that the application will allow. Zero, which is the default, indicates that all levels can be used.

D3DTSS_MINFILTER

Member of the **CONST_D3DTEXTUREMINFILTER** enumeration that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFN_POINT.

D3DTSS_MIPFILTER

Member of the **CONST_D3DTEXTUREMIPFILTER** enumeration that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFP_NONE.

D3DTSS_TEXCOORDINDEX

Index of the texture coordinate set to use with this texture stage. The default index is 0. Set this state to the zero-based index of the texture set at for each vertex that this texture stage will use. (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 using the u, v coordinates (0,0).

You can include flags from the **CONST_D3DTEXCOORDINDEXFLAGS** enumeration with the index you specify for this state to modify how the system interprets the index value. For more information, see *Configuring Automatically Generated Texture Coordinates*.

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.

CONST_D3DTEXTURETRANSFORMFLAGS

The **CONST_D3DTEXTURETRANSFORMFLAGS** enumeration is used with the D3DTSS_TEXTURETRANSFORMFLAGS texture stage state.

IDH_CONST_D3DTEXTURETRANSFORMFLAGS_d3d_vb

Type CONST_D3DTEXTURETRANSFORMFLAGS

D3DTTFF_COUNT1 = 1
 D3DTTFF_COUNT2 = 2
 D3DTTFF_COUNT3 = 3
 D3DTTFF_COUNT4 = 4
 D3DTTFF_DISABLE = 0
 D3DTTFF_PROJECTED = 256

End Type

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_DISABLE

Texture coordinates are to be passed directly to the rasterizer.

D3DTTFF_PROJECTED

The texture coordinates should all be 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 will be divided by the third coordinate before being passed to the rasterizer.

See Also

D3DTSS_TEXTURETRANSFORMFLAGS

CONST_D3DTRANSFORMSTATETYPE

The **CONST_D3DTRANSFORMSTATETYPE** enumeration defines values that describe the transformation state.

Enum CONST_D3DTRANSFORMSTATETYPE

D3DTRANSFORMSTATE_WORLD = 1
 D3DTRANSFORMSTATE_WORLD1 = 4,
 D3DTRANSFORMSTATE_WORLD2 = 5,
 D3DTRANSFORMSTATE_WORLD3 = 6,
 D3DTRANSFORMSTATE_VIEW = 2
 D3DTRANSFORMSTATE_PROJECTION = 3
 D3DTRANSFORMSTATE_TEXTURE0 = 16

IDH_CONST_D3DTRANSFORMSTATETYPE_d3d_vb

```

D3DTRANSFORMSTATE_TEXTURE1 = 17
D3DTRANSFORMSTATE_TEXTURE2 = 18
D3DTRANSFORMSTATE_TEXTURE3 = 19
D3DTRANSFORMSTATE_TEXTURE4 = 20
D3DTRANSFORMSTATE_TEXTURE5 = 21
D3DTRANSFORMSTATE_TEXTURE6 = 22
D3DTRANSFORMSTATE_TEXTURE7 = 23
End Enum

```

D3DTRANSFORMSTATE_WORLD, D3DTRANSFORMSTATE_VIEW, and D3DTRANSFORMSTATE_PROJECTION

Define the matrices for the world, view, and projection transformations. The default values are Nothing (the identity matrices).

D3DTRANSFORMSTATE_WORLD1, D3DTRANSFORMSTATE_WORLD2, D3DTRANSFORMSTATE_WORLD3

Identify 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.

D3DTRANSFORMSTATE_TEXTURE0 through D3DTRANSFORMSTATE_TEXTURE7

Identifies the transformation matrix being set for the specified texture stage.

See Also

CONST_D3DRENDERSTATETYPE, Geometry Blending

CONST_D3DVBCAPSFLAGS

The **CONST_D3DVBCAPSFLAGS** enumeration defines vertex buffer capability flags that are used in the **ICaps** member of the **D3DVERTEXBUFFERDESC** type.

```

Enum CONST_D3DVBCAPSFLAGS
    D3DVBCAPS_DEFAULT = 0
    D3DVBCAPS_DONOTCLIP = 1
    D3DVBCAPS_OPTIMIZED = 2147483648
    D3DVBCAPS_SYSTEMMEMORY = 2048
    D3DVBCAPS_WRITEONLY = 65536
End Enum

```

D3DVBCAPS_DEFAULT

The vertex buffer should be created in whatever memory the driver chooses to allow efficient read operations.

D3DVBCAPS_DONOTCLIP

The vertex buffer cannot contain clipping information.

IDH_CONST_D3DVBCAPSFLAGS_d3d_vb

D3DVBCAPS_OPTIMIZED

The vertex buffer contains optimized vertex data. (This flag is not used when creating a new vertex buffer.)

D3DVBCAPS_SYSTEMMEMORY

The vertex buffer should be created in system memory. Use this capability for vertex buffers that will be rendered by using software devices (MMX and RGB devices).

D3DVBCAPS_WRITEONLY

Hints to the system that the application will only write 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.

CONST_D3DVERTEXBLENDFLAGS

The **CONST_D3DVERTEXBLENDFLAGS** enumeration defines flags used to control the number of matrices that the system applies when performing geometry blending. Members of this enumeration are used with the **D3DRENDERSTATE_VERTEXBLEND** render state.

Enum **CONST_D3DVERTEXBLENDFLAGS**

D3DVBLEND_1WEIGHT = 1

D3DVBLEND_2WEIGHTS = 2

D3DVBLEND_3WEIGHTS = 3

D3DVBLEND_DISABLE = 0

End Enum

D3DVBLEND_1WEIGHT

Enable vertex blending between the two matrices set by the **D3DTRANSFORMSTATE_WORLD** and **D3DTRANSFORMSTATE_WORLD1** transformation states.

D3DVBLEND_2WEIGHTS

Enable vertex blending between the three matrices set by the **D3DTRANSFORMSTATE_WORLD**, **D3DTRANSFORMSTATE_WORLD1**, and **D3DTRANSFORMSTATE_WORLD2** transformation states.

D3DVBLEND_3WEIGHTS

Enable vertex blending between the four matrices set by the **D3DTRANSFORMSTATE_WORLD**, **D3DTRANSFORMSTATE_WORLD1**, **D3DTRANSFORMSTATE_WORLD2**, and **D3DTRANSFORMSTATE_WORLD3** transformation states.

D3DVBLEND_DISABLE

Disable vertex blending; only apply the world matrix specified by the **D3DTRANSFORMSTATE_WORLD** transformation state.

IDH_CONST_D3DVERTEXBLENDFLAGS_d3d_vb

Remarks

Geometry blending (multi-matrix vertex blending) requires that your application use a vertex format that has blending (beta) weights for each vertex.

See Also

CONST_D3DRENDERSTATETYPE,
CONST_D3DTRANSFORMSTATETYPE, **Direct3DDevice7.SetTransform**,
Geometry Blending

CONST_D3DVOPFLAGS

The **CONST_D3DVOPFLAGS** enumeration devices vertex operation flags used in the *vertexOp* parameter of the **Direct3DVertexBuffer7.ProcessVertices** method.

```
Enum CONST_D3DVOPFLAGS
    D3DVOP_CLIP      = 2
    D3DVOP_EXTENTS   = 4
    D3DVOP_LIGHT     = 1024
    D3DVOP_TRANSFORM = 1
End Enum
```

D3DVOP_CLIP

Transform the vertices and clip any vertices that exist outside the viewing frustum. This flag cannot be used with vertex buffers that do not contain clipping information (for example, created with the **D3DDP_DONOTCLIP** flag).

D3DVOP_EXTENTS

Transform the vertices, then update the extents of the screen rectangle when the vertices are rendered. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice7.GetClipStatus** will not have been updated to account for the vertices when they are rendered.

D3DVOP_LIGHT

Light the vertices.

D3DVOP_TRANSFORM

Transform the vertices using the world, view, and projection matrices. This flag must always be set.

IDH_CONST_D3DVOPFLAGS_d3d_vb

CONST_D3DVTXPCAPS

The **CONST_D3DVTXPCAPS** enumeration defines flags that describe the vertex processing capabilities that the device exposes in the **IVertexProcessingCaps** member of the **D3DDEVICEDESC7** type.

Type **CONST_D3DVTXPCAPS**

D3DVTXPCAPS_DIRECTIONALLIGHTS = 8

D3DVTXPCAPS_MATERIALSOURCE7 = 2

D3DVTXPCAPS_NONLOCALVIEWER = 32

D3DVTXPCAPS_POSITIONALLIGHTS = 16

D3DVTXPCAPS_TEXGEN = 1

D3DVTXPCAPS_VERTEXFOG = 4

End Type

D3DVTXPCAPS_DIRECTIONALLIGHTS

Device supports directional lights.

D3DVTXPCAPS_MATERIALSOURCE7

Device supports selectable vertex color sources. For more information, see Light Color Types and Sources.

D3DVTXPCAPS_NONLOCALVIEWER

Device supports orthogonal specular highlights, enabled by setting the **D3DRENDERSTATE_LOCALVIEWER** render state to False.

D3DVTXPCAPS_POSITIONALLIGHTS

Device supports positional lights (including point lights and spotlights).

D3DVTXPCAPS_TEXGEN

Device can generate texture coordinates.

D3DVTXPCAPS_VERTEXFOG

Device supports vertex fog.

See Also

D3DDEVICEDESC7

CONST_D3DZBUFFERTYPE

The **CONST_D3DZBUFFERTYPE** enumeration describes depth-buffer formats for use with the **D3DRENDERSTATE_ZENABLE** render state.

Enum **CONST_D3DZBUFFERTYPE**

D3DZB_FALSE = 0

D3DZB_TRUE = 1

D3DZB_USEW = 2

IDH_CONST_D3DVTXPCAPS_d3d_vb

IDH_CONST_D3DZBUFFERTYPE_d3d_vb

End Enum

D3DZB_FALSE

Disable depth-buffering.

D3DZB_TRUE

Enable z-buffering.

D3DZB_USEW

Enable w-buffering.

Remarks

The D3DZB_FALSE and D3DZB_TRUE values are interchangeable with the True and False macro values previously used with D3DRENDERSTATE_ZENABLE.

See Also

Direct3DDevice7.SetRenderState, Depth Buffers

Flexible Vertex Format Flags

Direct3D Immediate Mode uses flag values to describe vertex formats used for DrawPrimitive-based rendering. The **CONST_D3DFVFFLAGS** enumeration defines the following flags to explicitly describe a vertex format, and provides helper macros that act as common combinations of such flags. For more information, see About Vertex Formats.

Flexible vertex format (FVF) flags

D3DFVF_DIFFUSE

Vertex format includes a diffuse color component.

D3DFVF_LVERTEX

Vertex format is equivalent to the **D3DLVERTEX** vertex type.

D3DFVF_NORMAL

Vertex format includes a vertex normal vector. This flag cannot be used with the D3DFVF_XYZRHW flag.

D3DFVF_SPECULAR

Vertex format includes a specular color component.

D3DFVF_TLVERTEX

Vertex format is equivalent to the **D3DTLVERTEX** vertex type.

D3DFVF_VERTEX

Vertex format is equivalent to the **D3DVERTEX** vertex type.

D3DFVF_XYZ

Vertex format includes the position of an untransformed vertex. This flag cannot be used with the D3DFVF_XYZRHW flag. If you use this flag, you must also specify a vertex normal, a vertex color component (D3DFVF_DIFFUSE or

D3DFVF_SPECULAR), or include at least one set of texture coordinates (D3DFVF_TEX1 through D3DFVF_TEX8).

D3DFVF_XYZRHW

Vertex format includes the position of a transformed vertex. This flag cannot be used with the D3DFVF_XYZ or D3DFVF_NORMAL flags. If you use this flag, you must also specify a vertex color component (D3DFVF_DIFFUSE or D3DFVF_SPECULAR) or include at least one set of texture coordinates (D3DFVF_TEX1 through D3DFVF_TEX8).

D3DFVF_XYZB1 through D3DFVF_XYZB5

Vertex format contains position data, and a corresponding number of weighting (beta) values to be used for multimatrix vertex blending operations. Currently, 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.

See Also

About Vertex Formats, Geometry Blending

Texture Argument Flags

Each texture stage for a device can have two texture arguments that affect the color or alpha channel of the texture. You set and retrieve texture arguments by calling the **Direct3DDevice7.SetTextureStageState** and

Direct3DDevice7.GetTextureStageState, 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 $s - 2$.)

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.

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 diffuse color, the default color is 0xFFFFFFFF.

D3DTA_TEXTURE

The texture argument is the texture color for this texture stage. This is valid only for the first color and alpha arguments in a stage (the D3DTSS_COLORARG1 and D3DTSS_ALPHAARG1 constants of **CONST_D3DTEXTURESTAGESTATETYPE**). 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.

D3DTA_TFACTOR

The texture argument is the texture factor set in a previous call to the **Direct3DDevice7.SetRenderState** with the D3DRENDERSTATE_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 such that, if the result of the argument were referred to by the variable x , the value would be $1.0 - x$.

Error Codes

Errors, defined by the **CONST_D3DIMERR** enumeration, are represented by negative values and cannot be combined. This table lists the error codes that can be generated by all Direct3D Immediate Mode methods. See the individual method descriptions for lists of the values each can return.

D3D_OK

No error occurred.

D3DERR_BADMAJORVERSION

The service you requested is unavailable in this major version of DirectX. (A "major version" denotes a primary release, such as DirectX 6.0.)

D3DERR_BADMINORVERSION

The service you requested is available in this major version of DirectX, but not in this minor version. Get the latest version of the component runtime from Microsoft. (A "minor version" denotes a secondary release, such as DirectX 6.1.)

D3DERR_COLORKEYATTACHED

The application attempted to create a texture with a surface that uses a color key for transparency.

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 palettized textures simultaneously enabled also share the same palette.

D3DERR_CONFLICTINGRENDERSTATE

The currently set render states cannot be used together.

D3DERR_DEVICEAGGREGATED

The **Direct3DDevice7.SetRenderTarget** method was called on a device that was retrieved from the render target surface.

D3DERR_INITFAILED

A rendering device could not be created because the new device could not be initialized.

D3DERR_INBEGIN

The requested operation cannot be completed while scene rendering is taking place. Try again after the scene is completed and the **Direct3DDevice7.EndScene** method (or equivalent method) is called.

D3DERR_INBEGINSTATEBLOCK

The operation cannot be completed while recording states for a state block. Complete recording by calling the **Direct3DDevice7.EndStateBlock** method and try again.

D3DERR_INOVERLAYSTATEBLOCK

The operation cannot be completed while overlaying a state block. Remove the state block overlay and try again.

D3DERR_INVALID_DEVICE

The requested device type is not valid.

D3DERR_INVALIDCURRENTVIEWPORT

The currently selected viewport is not valid.

D3DERR_INVALIDMATRIX

The requested operation could not be completed because the combination of the currently set world, view, and projection matrices is invalid (the determinant of the combined matrix is zero).

D3DERR_INVALIDPALETTE

The palette associated with a surface is invalid.

D3DERR_INVALIDPRIMITIVETYPE

The primitive type specified by the application is invalid.

D3DERR_INVALIDRAMPTEXTURE

Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.

D3DERR_INVALIDSTATEBLOCK

The state block handle is invalid.

D3DERR_INVALIDVERTEXFORMAT

The combination of flexible vertex format flags specified by the application is not valid.

D3DERR_INVALIDVERTEXTYPE

The vertex type specified by the application is invalid.

D3DERR_LIGHT_SET_FAILED

The attempt to set lighting parameters for a light object failed.

D3DERR_LIGHTHASVIEWPORT

The requested operation failed because the light object is associated with another viewport.

D3DERR_LIGHTNOTINTHISVIEWPORT

The requested operation failed because the light object has not been associated with this viewport.

D3DERR_MATERIAL_CREATE_FAILED

The material could not be created. This typically occurs when no memory is available to allocate for the material.

D3DERR_MATERIAL_DESTROY_FAILED

The memory for the material could not be deallocated.

D3DERR_MATERIAL_GETDATA_FAILED

The material parameters could not be retrieved.

D3DERR_MATERIAL_SETDATA_FAILED

The material parameters could not be set.

D3DERR_MATRIX_CREATE_FAILED

The matrix could not be created. This can occur when no memory is available to allocate for the matrix.

D3DERR_MATRIX_DESTROY_FAILED

The memory for the matrix could not be deallocated.

D3DERR_MATRIX_GETDATA_FAILED

The matrix data could not be retrieved. This can occur when the matrix was not created by the current device.

D3DERR_MATRIX_SETDATA_FAILED

The matrix data could not be set. This can occur when the matrix was not created by the current device.

D3DERR_NOCURRENTVIEWPORT

The viewport parameters could not be retrieved because none have been set.

D3DERR_NOTINBEGIN

The requested rendering operation could not be completed because scene rendering has not begun. Call **Direct3DDevice7.BeginScene** to begin rendering then try again.

D3DERR_NOTINBEGINSTATEBLOCK

The requested operation could not be completed because it is only valid while recording a state block. Call the **Direct3DDevice7.BeginStateBlock** method and try again.

D3DERR_NOVIEWPORTS

The requested operation failed because the device currently has no viewports associated with it.

D3DERR_SCENE_BEGIN_FAILED

Scene rendering could not begin.

D3DERR_SCENE_END_FAILED

Scene rendering could not be completed.

D3DERR_SCENE_IN_SCENE

Scene rendering could not begin because a previous scene was not completed by a call to the **Direct3DDevice7.EndScene** method.

D3DERR_SCENE_NOT_IN_SCENE

Scene rendering could not be completed because a scene was not started by a previous call to the **Direct3DDevice7.BeginScene** method.

D3DERR_SETVIEWPORTDATA_FAILED

The viewport parameters could not be set.

D3DERR_STENCILBUFFER_NOTPRESENT

The requested stencil buffer operation could not be completed because there is no stencil buffer attached to the render target surface.

D3DERR_SURFACENOTINVIDMEM

The device could not be created because the render target surface is not located in video-memory. (Hardware-accelerated devices require video-memory render target surfaces.)

D3DERR_TEXTURE_BADSIZE

The dimensions of a current texture are invalid. This can occur when an application attempts to use a texture that has non-power-of-two dimensions with a device that requires them.

D3DERR_TEXTURE_CREATE_FAILED

The texture handle for the texture could not be retrieved from the driver.

D3DERR_TEXTURE_DESTROY_FAILED

The device was unable to deallocate the texture memory.

D3DERR_TEXTURE_GETSURF_FAILED

The DirectDraw surface used to create the texture could not be retrieved.

D3DERR_TEXTURE_LOAD_FAILED

The texture could not be loaded.

D3DERR_TEXTURE_LOCK_FAILED

The texture could not be locked.

D3DERR_TEXTURE_LOCKED

The requested operation could not be completed because the texture surface is currently locked.

D3DERR_TEXTURE_NO_SUPPORT

The device does not support texture mapping.

D3DERR_TEXTURE_NOT_LOCKED

The requested operation could not be completed because the texture surface is not locked.

D3DERR_TEXTURE_SWAP_FAILED

The texture handles could not be swapped.

D3DERR_TEXTURE_UNLOCK_FAILED

The texture surface could not be unlocked.

D3DERR_TOOMANYOPERATIONS

The application is requesting more texture filtering operations than the device supports.

D3DERR_TOOMANYPRIMITIVES

The device is unable to render the provided quantity of primitives in a single pass.

D3DERR_UNSUPPORTEDALPHAARG

The device does not support one of the specified texture blending arguments for the alpha channel.

D3DERR_UNSUPPORTEDALPHAOPERATION

The device does not support one of the specified texture blending operations for the alpha channel.

D3DERR_UNSUPPORTEDCOLORARG

The device does not support the one of the specified texture blending arguments for color values.

D3DERR_UNSUPPORTEDCOLOROPERATION

The device does not support the one of the specified texture blending operations 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_VBUF_CREATE_FAILED

The vertex buffer could not be created. This can happen when there is insufficient memory to allocate a vertex buffer.

D3DERR_VERTEXBUFFERLOCKED

The requested operation could not be completed because the vertex buffer is locked.

D3DERR_VERTEXBUFFEROPTIMIZED

The requested operation could not be completed because the vertex buffer is optimized. (The contents of optimized vertex buffers are driver specific, and considered private.)

D3DERR_VERTEXBUFFERUNLOCKFAILED

The vertex buffer could not be unlocked because the vertex buffer memory was overrun. Make sure that your application does not write beyond the size of the vertex buffer.

D3DERR_VIEWPORTDATANOTSET

The requested operation could not be completed because viewport parameters have not yet been set. Set the viewport parameters by calling **Direct3DDevice7.SetViewport** method and try again.

D3DERR_VIEWPORTHASNODEVICE

The requested operation could not be completed because the viewport has not yet been associated with a device.

D3DERR_WRONGTEXTUREFORMAT

The pixel format of the texture surface is not valid.

D3DERR_ZBUFF_NEEDS_SYSTEMMEMORY

The requested operation could not be completed because the specified device requires system-memory depth-buffer surfaces. (Software rendering devices require system-memory depth buffers.)

D3DERR_ZBUFF_NEEDS_VIDEOMEMORY

The requested operation could not be completed because the specified device requires video-memory depth-buffer surfaces. (Hardware-accelerated devices require video-memory depth buffers.)

D3DERR_ZBUFFER_NOTPRESENT

The requested operation could not be completed because the render target surface does not have an attached depth buffer.

Direct3D Immediate Mode Tools and Samples

This section describes sample applications included with the Microsoft® DirectX® SDK that demonstrate the use of Microsoft® Direct3D® Immediate Mode. Descriptions are organized as follows:

- Direct3D Immediate Mode Tools
- Direct3D Immediate Mode C++ Samples
- Direct3D Immediate Mode Visual Basic Samples

Direct3D Immediate Mode Tools

This section describes the following tool supplied with the DirectX Software Development Kit (SDK):

- MFCTex Tool

MFCTex Tool

Description

MFCTex is an interactive tool that lets you experiment with multiple texture blending and see the results instantly. The tool also generates the necessary code for achieving the desired effect.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Mfctex

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Mfctex.exe

User's Guide

Select up to three bitmaps, and change the parameters by using the text boxes and drop-down lists. To use fewer than three states, leave one of the **Texture File** text boxes blank.

When you have achieved an effect that you want to use in your application, click **View Code**, and copy all the text from the **Direct3D Texture Stage Code** dialog box.

For information on the parameters that you can change, see Multiple Texture Blending.

Direct3D Immediate Mode C++ Samples

[\[Visual Basic\]](#)

This section pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

This section provides descriptions of sample applications provided with the DirectX SDK that are primarily intended to demonstrate how to use Direct3D Immediate Mode. The following samples are described:

- Bend Sample
- Billboard Sample
- Boids Sample
- BumpEarth Sample
- BumpWaves Sample
- ClipMirror Sample
- Compress Sample

- D3DFrame Library
- Dolphin Sample
- EnumTex Sample
- EnvCube Sample
- Filter Sample
- Fireworks Sample
- Flare Sample
- LightMap Sample
- Lights Sample
- MFCFog Sample
- MipMap Sample
- MTexture Sample
- ScreenSaver Sample
- ShadowVol Sample
- ShadowVol2 Sample
- Spheremap Sample
- StencilDepth Sample
- StencilMirror Sample
- VBuffer Sample
- VideoTex Sample
- WBuffer Sample
- XFile Sample

Note

Most of the sample applications have a common interface (found in the D3DFrame Library) that has no menu bar. In addition to normal Microsoft® Windows® functions (minimize, maximize, restore, resize, and close), the interface supports the following commands:

Key/input	Command
ESC	Quit
F1	Help/About dialog box (lists these keystrokes)
F2	Device options
ALT+ENTER	Toggle between full-screen and windowed mode
Right mouse button	Display options popup menu.

The device options dialog box allows the user to change the driver, Direct3D device, or display mode (full-screen only) at run time. Not all devices can support rendering in a window at all display depths. Three-dimensional (3-D) hardware has finite video

memory that may be exceeded for certain display modes and window sizes. In either case, the samples display a message and default to a software rasterizer.

Although 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

[Direct3D Immediate Mode C/C++ Tutorials](#)

Bend Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See [Direct3D Immediate Mode Visual Basic Samples](#).

[\[C++\]](#)

Description

The Bend sample demonstrates a technique called surface skinning. It displays a 3-D object, which rotates about the y-axis and appears to bend.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Bend

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Bend.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

The sample achieves the surface-skinning effect by using two static copies of the object, one of which is oscillating along an axis. At each frame, the vertices of the two objects are merged and blended into a third object. The sample displays the object derived from the third set of vertices.

This sample was built using the Direct3D sample framework.

Billboard Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The Billboard sample demonstrates the billboarding technique. Billboarding is a way of making two-dimensional (2-D) sprites appear to be 3-D. It can also be used for smoke, clouds, vapor trails, energy blasts, and so on. For more information, see Common Techniques and Special Effects.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Billboard

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Billboard.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

The sample displays a grassy field with trees in it. The trees look like 3-D objects. However, they are actually 2-D texture bitmaps that are blended onto invisible rectangular polygons.

As the sample runs, the viewpoint changes. Each time it does, all the billboard polygons that the trees are painted onto are rotated so that they face the viewer. The sample then blends the images of the trees onto the billboard polygons. The trees appear to be 3-D because they can be viewed from all angles. However, close inspection reveals that the trees have exactly the same appearance from all angles. For many applications, this minor drawback is not noticeable.

The shadows are also 2-D textures.

This sample was built using the Direct3D sample framework.

Boids Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[C++]

Description

The Boids sample shows how to write a Direct3D application. Using a flocking algorithm, the sample moves a group of 3-D objects over a simple landscape.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Boids

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Boids.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

The Boids sample demonstrates the fundamentals of creating a 3-D environment and animating a group of objects in it.

BumpEarth Sample

[Visual Basic]

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[C++]

Description

The BumpEarth sample demonstrates the bump-mapping capabilities of Direct3D. Bump mapping is a texture-blending technique used to render the appearance of rough surfaces.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\BumpEarth

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\BumpEarth.exe

User's Guide

Press F1 to see available keystroke commands, or choose from the menu.

Your graphics hardware might not support bump mapping; if not, Direct3D displays a message to that effect when you attempt to run this sample. The solution is to enable the reference rasterizer by running (*SDK root*)
\Samples\Multimedia\D3dim\Bin\Enablerefrast.reg.

Programming Notes

Bump mapping is an advanced multitexture blending technique that can be used to render the appearance of rough surfaces. The bump map itself is a texture that stores the perturbation data.

In this sample, the map of the world is a texture. The sample blends both the map texture and the bump-map texture onto the sphere to give the appearance of a high-resolution topographical map.

For more information, see Bump Mapping and the BumpWaves Sample.

This sample was built using the Direct3D sample framework.

BumpWaves Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The BumpWaves sample simulates reflections off waves, using bump mapping.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\BumpWaves

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\BumpWaves.exe

User's Guide

Press F1 to see available keystroke commands, or choose from the menu.

Your graphics hardware might not support bump mapping; if not, Direct3D displays a message to that effect when you attempt to run this sample. The solution is to enable the reference rasterizer. You can do so by running (*SDK root*)
\Samples\Multimedia\D3dim\Bin\Enablerefrast.reg.

Programming Notes

For more information, see Bump Mapping and the BumpEarth Sample.

The sample was built using the Direct3D sample framework.

ClipMirror Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The ClipMirror sample shows how to use clip planes to implement a planar mirror.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\ClipMirror

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\ClipMirror.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

The scene is reflected in a mirror and rendered in a second pass. The corners of the mirrors, together with the camera eye point, are used to define a custom set of clip planes so that the reflected geometry appears only within the mirror's boundaries.

Compress Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The Compress sample shows how to load the DDS file format into a compressed texture surface. DDS textures can be created using DxTex included with the DirectDraw tools in the DirectX SDK.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Compress

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Compress.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

The file format is called DDS because it encapsulates the information in a DirectDrawSurface. The data can be read directly into a surface of a matching format.

The ReadDDSTexture function demonstrates how a DDS surface is read from a file.

A DDS file has the following format:

DWORD dwMagic	(0x20534444, or "DDS ")
DDSURFACEDESC2 ddsd	Information about the surface format
BYTE bData1[]	Data for the main surface
[BYTE bData2[]]	Data for attached surfaces, if any, follows.

This format is easy to read and write, and it supports features such as alpha and multiple MIP levels, as well as DXTn compression. If it uses DXTn compression, it can be one of five compressed types. See Compressed Texture Surfaces.

After the texture is read in, a pixel format must be chosen that is supported by the renderer. In the sample, the supported pixel formats are enumerated and stored in a linked list. After the pixel formats are collected, the list is searched for a best match, using the **FindBestPixelFormatMatch** function.

Some Direct3D devices, such as the reference rasterizer and some hardware devices, can render compressed textures directly. For renderers that do not directly support this, the compressed surface must be blitted to a noncompressed surface. The **BltToUncompressedSurface** function shows how this is done.

D3DFrame Library

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The D3D framework is a set of C++ classes used to create the Direct3D Immediate Mode samples. It is not a part of the Direct3D API. The framework was created and used to provide consistency and clarity of presentation for the Direct3D samples. The framework classes may or may not be appropriate for use in your applications.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\D3DFrame

Executable: None.

User's Guide

D3DFrame compiles as a static library, which is used to build the remainder of the Direct3D Immediate Mode samples.

Programming Notes

The framework consists of classes that enumerate the DirectDraw drivers, Direct3D devices, and display modes available to each device. The samples use them to initialize and run Direct3D. The classes also help provide a consistent user interface for the set of samples. In addition, the framework includes a set of classes for loading and managing textures.

The Direct3D framework also contains numerous macros and functions for debugging, for manipulating Direct3D objects, and for doing math operations common in Direct3D programming.

Dolphin Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The Dolphin sample is a simple 3-D application that uses an animated model and textures.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Dolphin

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Dolphin.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

The dolphin is loaded as three separate meshes, which are linearly interpolated to produce the final mesh.

EnumTex Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The EnumTex sample demonstrates the recommended way of enumerating texture formats.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Dolphin

Executable: None.

User's Guide

No executable is included with the SDK because this sample does nothing visible.

Programming Notes

It is important to enumerate textures correctly so that applications remain compatible as new texture formats become available. The sample code shows how to do this.

EnvCube Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

This sample shows how to implement cube mapping, which is a new environment-mapping technique implemented in DirectX 7.0.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\EnvCube

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\EnvCube.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

A cube map is a special texture surface created by rendering a scene in six orthogonal directions. It is then used for traditional environment-mapping effects.

Filter Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The Filter sample demonstrates the texture-filtering techniques that Direct3D supports. Direct3D texture-filtering techniques enable applications to achieve greater realism in the appearance of rendered primitives. For more information, see Texture Filtering.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Filter

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Filter.exe

User's Guide

When the sample begins, it displays two rectangular primitives with textures on them.

In addition to the usual commands listed in the **About** dialog box when you press F1, this sample has a main menu.

The **File** menu contains choices for pausing and resuming the sample, changing the Direct3D device, and exiting. You can also pause and resume the sample by pressing the ENTER key.

The **Left Pane** menu controls the texture-filtering methods that the sample uses for magnification and reduction of the 3-D primitive on the left side of the screen. It also has options for edge antialiasing and anisotropic texture filtering. You might have to enable the software reference rasterizer to view the effects of these options. To enable the reference rasterizer, run (*SDK root*)\Samples\Multimedia\D3dim\Bin\Enablerefrast.reg.

The **Right Pane** menu allows you to set the filtering methods that the sample uses to perform magnification and reduction when it renders the primitive on the right side of the screen.

Programming Notes

This sample demonstrates nearest-point sampling, linear filtering, and anisotropic texture filtering. It shows how to enable and disable anisotropy. In addition, this sample shows how to enable edge antialiasing.

Fireworks Sample

[Visual Basic]

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The Fireworks sample implements a system of particles simulating a fireworks explosion. Particles are popular in games to show effects like smoke, sparks, and explosions.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Fireworks

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Fireworks.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

The fireworks explosion is simulated by using a system of particles; each particle is rendered with a partially transparent texture map of a sphere. The position and color of each particle are governed by parameterized equations and are updated in each frame.

Flare Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The Flare sample shows how to create a lens flare effect, using alpha blending.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Flare

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Flare.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

In this sample, lens flare is simulated with mathematical functions that govern the position of each flare. The flares are rendered using additive alpha blending and animated to give a sparkle effect.

LightMap Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

This sample shows how to use multitexturing and multipass techniques to create some complex lighting effects. It shows a light swinging in a room, which dynamically lights up the walls and ceiling as the light moves.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\LightMap

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Lightmap.exe

User's Guide

On the **Options** menu, choose between multipass and multiple-texture blending. The latter is not available if your hardware does not support it.

Press F1 to see other available commands, or choose from the menu.

Programming Notes

There is no true lighting in this sample—everything is done with light maps. Light maps are extremely useful in games because they are much faster than real lighting. In addition, real lighting is calculated only at the vertices, so highly tessellated meshes are required.

Lights Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[C++]

Description

The Lights sample shows how to turn on and move the various types of Direct3D lights. The lights fly around, illuminating the scene, and switch every few seconds to a new type of light.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Lights

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Lights.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

This sample shows how to set up the light properties for each of the types of lights available. It also shows how to dynamically orient the lights on each frame.

MFCFog Sample

[Visual Basic]

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[C++]

Description

The MFCFog sample does a fly-by over some terrain and allows you to see the effects of various fog parameters.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\MFCFog

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Mfcfog.exe

User's Guide

Change parameters by using the controls. For information on the parameters, see Fog. The effects are most striking in full-screen mode.

Programming Notes

Games that use terrain typically turn on fog so that they can prevent the rendering of objects in the far distance.

MipMap Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

This sample shows two brick walls moving back and forth along the z-axis. One uses mipmapping, and the other does not, giving a side-by-side comparison of the benefits of mipmapping.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\MipMap

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Mipmap.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

This sample does not use the texture code of D3DFrame, but shows the full implementation needed to load and build mipmapped textures.

MTexture Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The MTexture sample shows how to use multitexturing. The scene consists of a room with walls that have a base texture and a spotlight texture, each using a different set of texture coordinates.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\MTexture

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Mtexture.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

This sample shows how to program the multitexture stages, using the **IDirect3DDevice7::SetTextureStageState** method. Dozens of different effects are possible with this method. The sample shows one, monochrome light mapping, which is very popular in current game titles.

ScreenSaver Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The ScreenSaver sample shows how to create a Direct3D screen saver.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\ScreenSaver

Executable: None

User's Guide

When you compile the code, the output file is D3DBend.scr. You can view the screen saver by double-clicking on this file. To make it available in the Windows Display Properties dialog box, copy it to the \Windows\System directory.

Programming Notes

The startup code is in Scrnsave.lib.

For more information about compiling screen savers, see the Screen Saver Library topic in the Platform SDK User Interface Services documentation.

ShadowVol Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The ShadowVol sample shows how to create and use stencil buffers to implement shadow volumes. With shadow volumes, an object of any shape can cast a shadow onto another object of any shape.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\ShadowVol

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Shadowvol.exe

User's Guide

This sample runs only on devices that support stencil buffers.

Press F1 to see available commands, or choose from the menu.

Programming Notes

Shadow volumes are a fairly advanced technique.

To create shadow volumes

1. Build a set of polygonal faces that encompass the volume of the shadow.
 2. Use the stencil buffer to render the front-facing planes of the shadow volume.
 3. Set up the stencil buffer to render the back-facing planes, this time subtracting values from the stencil buffer. The stencil buffer then contains a mask of the cast shadow.
 4. Draw a large gray or black rectangle, using the stencil buffer as a mask, and the frame buffer will be updated with the shadow.
-

ShadowVol2 Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The ShadowVol2 sample shows how to create and use stencil buffers to implement shadow volumes, which are used to cast shadows on objects of any complexity. It is an extension of the ShadowVol Sample.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\ShadowVol2

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Shadowvol2.exe

User's Guide

This sample runs only on devices that support stencil buffers.

Press F1 to see available commands, or choose from the menu.

The following options are available on the **Shadow Modes** menu:

- **Draw Shadows:** Check this option to enable shadow rendering.
- **Show Shadow Volumes:** Instead of shadows, draw the shadow volumes used to compute them.
- **Draw Shadow Volume Caps:** If this is turned on, the end surfaces of the shadow volumes are drawn.
- **1-Bit Stencil Mode:** Use a different algorithm that uses only a 1-bit stencil buffer and in which overlapping shadows are not allowed. If the device supports only a 1-bit stencil, you cannot switch out of this mode.

- **Z-Order Shadow Vols:** In 1-bit mode, shadow volumes must be rendered front to back, so rendering might be incorrect unless this option is checked.

Programming Notes

Shadow volumes are a technique for casting shadows onto arbitrary nonplanar surfaces. The effect is achieved by constructing a shadow volume with respect to the light source and the object that casts the shadow. In this sample, the light source is a directional light whose direction circles about points on the plane, and the shadow volume is computed by projecting the vertices of the object that casts the shadow onto a plane perpendicular to the light, finding the 2-D convex hull of these points in the plane and extruding the 2-D hull in the light direction to form the 3-D shadow volume. The shadow volume must extend far enough that it covers any geometry that is in shadow. This particular shadow-volume computation requires that the shadow caster be a convex object.

The rendering proceeds as follows:

1. The geometry is rendered as normal.
 2. The shadow volume is rendered without writing to the z or color buffer.
Alpha blending is used here to avoid writing to the color buffer. Every place the shadow volume appears is marked in the stencil buffer.
 3. The culling order is reversed, and the back faces of the shadow volume are rendered, this time unmarking all the pixels that are covered in the stencil buffer.
These have passed the z-test, and thus are visible behind the back of the shadow volume; they are not in shadow. The pixels still marked are those that lie inside the front and back bounds of the shadow volume and are thus in shadow.
 4. The pixels that are still marked are blended with a large black rectangle that covers the viewport, generating the shadow.
-

Spheremap Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

This sample loads a 3-D object and renders it, using a sphere map.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\Spheremap

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Spheremap.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

The sphere map itself is a special, preconstructed texture map containing a 180-degree view of an environment. Before a frame is rendered, the object's normals are used to compute the texture coordinates for each vertex of the object. When rendered, the object looks as if it reflects the environment.

StencilDepth Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The StencilDepth sample shows how to use a stencil buffer to visualize the depth complexity of a scene.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\StencilDepth

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\StencilDepth.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

StencilMirror Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[C++]

Description

The StencilMirror sample shows how to use a stencil buffer to implement planar mirrors of any shape in a scene.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\StencilMirror

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\StencilMirror.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

VBuffer Sample

[Visual Basic]

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[C++]

Description

This sample shows how to use vertex buffers.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\VBuffer

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\VBuffer.exe

User's Guide

Press F1 to see available commands, or choose from the menu.

Programming Notes

Before vertex buffers, geometry rendering was accomplished by using an array of vertices passed to the **DrawPrimitive** function. Vertex buffers are more useful, but need to be specifically created, locked, filled, and unlocked before being rendered with the new **DrawPrimitiveVB** function.

VideoTex Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The VideoTex sample shows how to use an .avi file as a texture map.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\VideoTex

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\Videotex.exe

User's Guide

Press F1 to see a list of the usual commands, or choose from the menu.

Programming Notes

The sample draws a cube with an .avi texture mapped to each of its faces. The texture's surface must use the DDSCAPS2_HINTDYNAMIC flag.

WBuffer Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

The WBuffer sample shows how to use w-buffering.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\WBuffer

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\WBuffer.exe

User's Guide

Press F1 to see a list of the usual commands, or choose from the menu. In addition, you can change the buffering mechanism by pressing W (w-buffering, if supported), Z (z-buffering), or N (no buffering). Note the artifacts that appear when using z-buffering.

Programming Notes

W-buffering is a depth-buffering alternative to z-buffering and should be used in cases where z-buffering produces artifacts. W-buffering quantizes the depth buffer much better.

XFile Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C++. See Direct3D Immediate Mode Visual Basic Samples.

[\[C++\]](#)

Description

This sample shows how to load and render .x files.

Path

Source: (*SDK root*)\Samples\Multimedia\D3dim\Src\XFile

Executable: (*SDK root*)\Samples\Multimedia\D3dim\Bin\XFile.exe

User's Guide

When you run the sample, an **Open File** dialog box appears. You can find some .x files in the media folder of the D3DIM samples directory. If there is a .bmp file of the same name in the directory, it is automatically used as a texture for the object.

Press F1 to see available commands. In addition, you can load a different file from the **File** menu.

Programming Notes

If a texture appears upside-down on an object that you have loaded, it is because of the way the object loads the texture. An earlier version of Direct3D loaded bitmaps upside-down, so the creators of some .x files compensated by reversing the coordinates.
