Direct3D Immediate-Mode: Overview

This section provides overview information about the Direct3D® Immediate Mode. Information is divided into the following groups:

- About Direct3D Immediate Mode
- Why Use Direct3D Immediate Mode?
- Getting Started with Immediate Mode
- Direct3D Immediate-Mode Architecture
- Direct3D Immediate-Mode Essentials
- Direct3D Execute-Buffer Tutorial

About Direct3D Immediate Mode

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

You can use Direct3D in either of two modes: Immediate Mode or Retained Mode. Retained Mode is a high-level 3-D application programmer interface (API) for programmers who require rapid development or who want the help of Retained Mode's built-in support for hierarchies and animation.

Microsoft developed the Direct3D Immediate Mode as a low-level 3-D API. Immediate Mode is ideal for developers who need to port games and other high-performance multimedia applications to the Microsoft 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 z-buffering
- Flat and Gouraud shading
- Phong lighting model, with multiple lights and light types
- Full material and texture support, including mipmapping
- Ramp and RGB software emulation
- Transformation and clipping
- Hardware independence

- Full support on NT
- 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 probably the sample code included with this Software Development Kit (SDK); it illustrates how to put Direct3D Immediate Mode to work in real-world applications.

This section is not an introduction to programming with Direct3D Immediate Mode; for this information, see Direct3D Execute-Buffer Tutorial.

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 and provides emulation for missing hardware functionality. (The programmer is always told which capabilities are in hardware and which are being emulated.) 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.

There are two ways to use Immediate Mode: you can use the DrawPrimitive methods or you can work with execute buffers (display lists). Most developers who have never worked with Immediate Mode before will use the DrawPrimitive methods. Developers who already have an investment in code that uses execute buffers will probably continue to work with them. Neither technique is faster than the other — which you choose will depend on the needs of your application and your preferred programming style. For more information about these two ways to work with Immediate Mode, see The DrawPrimitive Methods and Using Execute Buffers.

Immediate Mode allows a low-overhead connection to 3-D hardware; there is no data translation between Direct3D and the Direct3D hardware-abstraction layer (HAL) for rendering operations. 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 you need to understand before you write programs that incorporate 3-D graphics. This is not a discussion of broad architectural details, nor is it an in-depth analysis of specific Direct3D components. (For information about these topics, see Direct3D Immediate-Mode Architecture and Direct3D Immediate-Mode Essentials.)

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

Information in this section is divided into the following groups:

- 3-D Coordinate Systems
- 3-D Transformations
- Polygons
- Triangle Strips and Fans
- Triangle Rasterization Rules

3-D Coordinate Systems

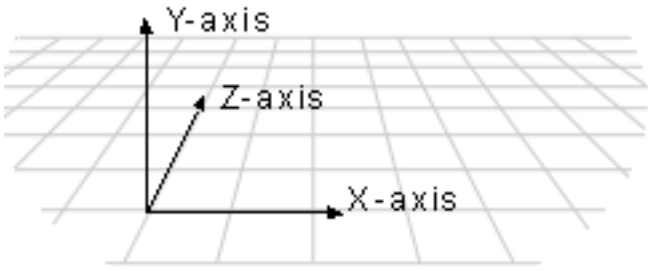
This section describes the Direct3D coordinate system and coordinate types that your application can use.

- Direct3D Coordinate System
- U- and V-Coordinates

Direct3D Coordinate System

There are two varieties of Cartesian coordinate systems in 3-D graphics: 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. This means the positive z-axis points away from the viewer, as shown in the following illustration:



In a left-handed coordinate system, rotations occur clockwise around any axis that is pointed at the viewer.

If you need to work in a right-handed coordinate system — for example, if you are porting an application that relies on right-handedness — you can do so by making two simple changes to the data passed to Direct3D.

- Flip the order of triangle vertices so that the system traverses them clockwise from the front. In other words, if the vertices are v0, v1, v2, pass them to Direct3D as v0, v2, v1.
- Scale the projection matrix by -1 in the z-direction. To do this, flip the signs of the _13, _23, _33, and _43 members of the **D3DMATRIX** structure.

U- and V-Coordinates

In addition to the x-, y-, and z-coordinates that define space in a Cartesian coordinate system, Direct3D uses texture coordinates. These coordinates (u and v) define an "up" direction for the texture, typically along the y-axis (u) and an orientation along the plane of the texture, typically along the z-axis (v). As with all normalized vectors, the origins of these vectors are at [0,0,0].

For more information about texture coordinates, see Textures.

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, shear, and size objects.
- Change viewing positions, directions, and perspective.

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'):

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

You perform the following operations on (x, y, z) and the matrix to produce the point (x', y', z'):

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

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.

Matrices are specified in row order. For example, the following matrix could be represented by an array:

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

The array for this matrix would look like this:

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

This section describes the 3-D transformations available to your applications through Direct3D:

- Translation
- Rotation
- Scaling

For more information about transformations in Direct3D Immediate Mode, see Viewports and Transformations.

Translation

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

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

Rotation

The transformations described in this section are for left-handed coordinate systems, and so may be different from transformation matrices you have seen elsewhere.

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

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

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

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

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

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

Note that in these example matrices, the Greek letter theta stands for the angle of rotation, specified in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

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'):

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

Polygons

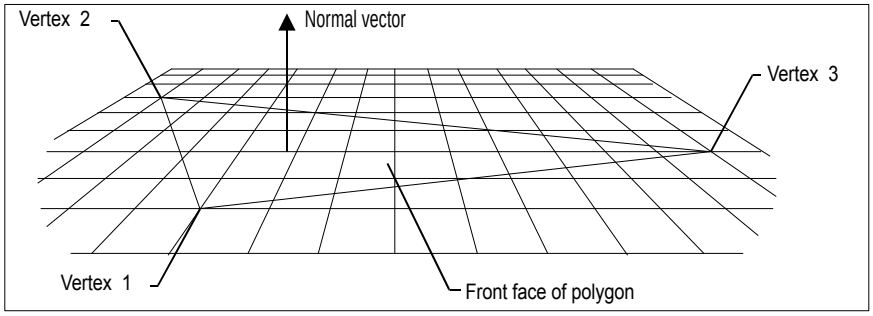
Three-dimensional objects in Direct3D are made up of meshes. A mesh is a set of faces, each of which is described by one or more triangles.

This section describes how your applications can use Direct3D polygons.

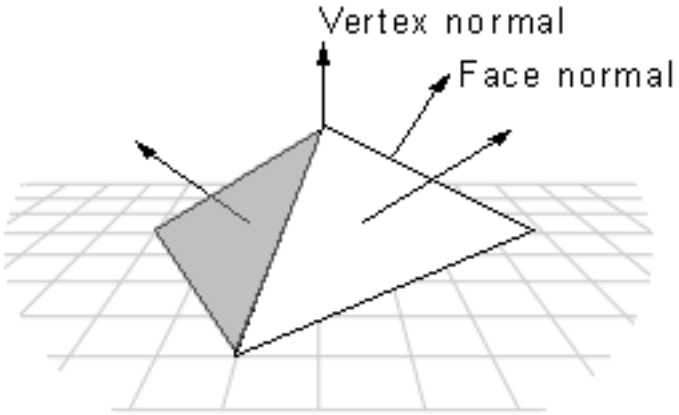
- Face and Vertex Normals
- Shade Modes
- Triangle Interpolants

Face and Vertex Normals

Each face in a mesh has a perpendicular normal vector whose direction is determined by the order in which the vertices are defined and by whether the coordinate system is right- or left-handed. If the normal vector of a face is oriented toward the viewer, that side of the face is its front. In Direct3D, only the front side of a face is visible, and a front face is one in which vertices are defined in clockwise order.



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 shade mode. For Gouraud shade modes, and for controlling lighting and texturing effects, the system uses vertex normals.

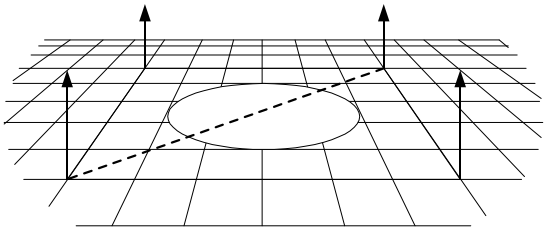


Throughout Direct3D, vertices describe position and orientation. Each vertex in a primitive is described by a vector that gives its position and a normal vector that gives its orientation, texture coordinates, and a color.

Shade Modes

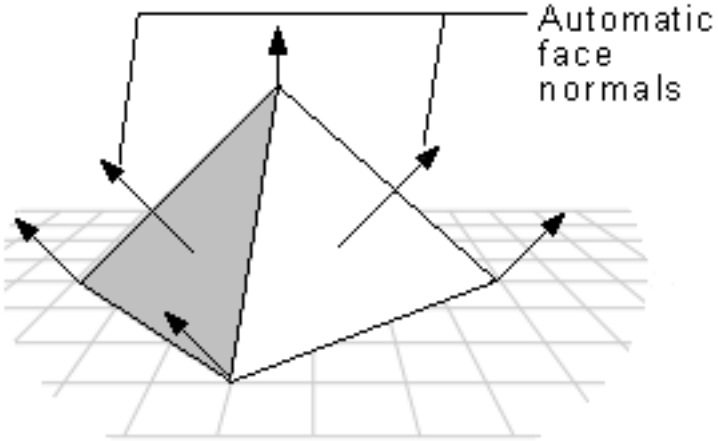
In the flat shade mode, the system duplicates the color of one vertex across all the other faces of the primitive. In the Gouraud and Phong shade modes, vertex normals are used to give a smooth look to a polygonal object. In Gouraud shading, the color and intensity of adjacent vertices is interpolated across the space that separates them. In Phong shading, which is not currently supported by Direct3D, the system calculates the appropriate shade value for each pixel on a face.

Most applications use Gouraud shading because it allows objects to appear smooth and is computationally efficient. Gouraud shading can miss details that Phong shading will not, however. For example, Gouraud and Phong shading would produce very different results in the case shown by the following illustration, in which a spotlight is completely contained within a face.

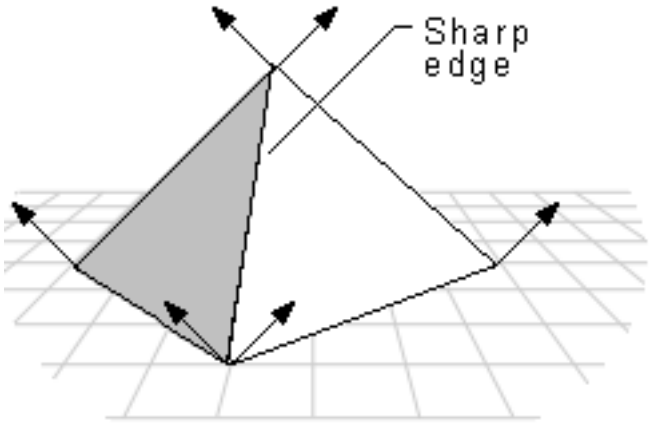


In this case, the Phong shade mode would calculate the value for each pixel and display the spotlight. The Gouraud shade mode, which interpolates between vertices, would miss the spotlight altogether; the face would be rendered as though the spotlight did not exist.

In the flat shade mode, the following pyramid would be displayed with a sharp edge between adjoining faces; the system would generate automatic face normals. In the Gouraud or Phong shade modes, however, shading values would be interpolated across the edge, and the final appearance would be of a curved surface.



If you want to use the Gouraud shade mode to display curved surfaces and you also want to include some objects with sharp edges, your application would need to duplicate the vertex normals at any intersection of faces where a sharp edge was required, as shown in the following illustration.



In addition to allowing a single object to have both curved and flat surfaces, the Gouraud shade mode lights flat surfaces more realistically than the flat shade mode. A face in the flat shade mode is a uniform color, but Gouraud shading allows light to fall across a face correctly; this effect is particularly obvious if there is a nearby point source. Gouraud shading is the preferred shade mode for most Direct3D applications.

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
- Fog
- Alpha

All of the triangle interpolants are modified by the current shade 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.
Phong	Vertex parameters are reevaluated for each pixel in the face, using the current lighting. The Phong shade mode is not currently supported.

The color and specular interpolants are treated differently, depending on the color model. In the RGB color model (**D3DCOLOR_RGB**), the system uses the red, green, and blue color components in the interpolation. In the monochromatic or "ramp" model (**D3DCOLOR_MONO**), the system uses only the blue component of the vertex color.

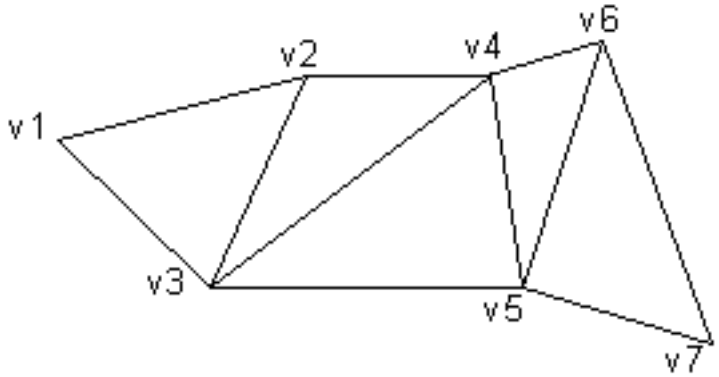
For example, if the red component of the color of vertex 1 were 0.8 and the red component of vertex 2 were 0.4, in the Gouraud shade mode and RGB color model the system would use interpolation to assign a red component of 0.6 to the pixel at the midpoint of the line between these vertices.

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.

An application can use the **dwShadeCaps** member of the **D3DPRIMCAPS** structure to determine what forms of interpolation the current device driver supports.

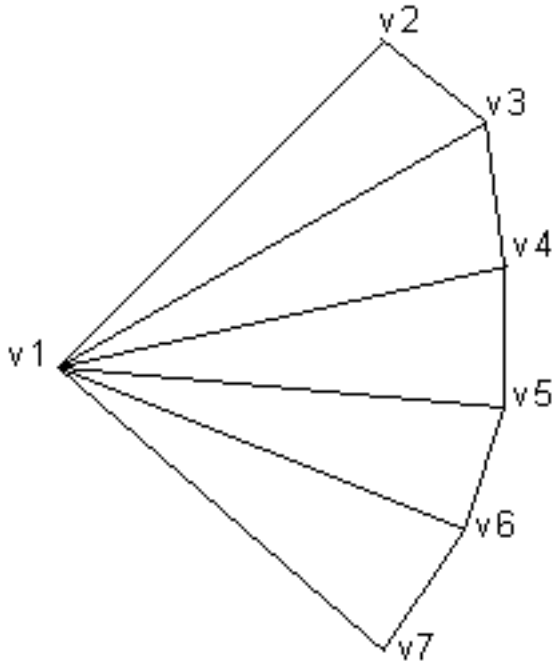
Triangle Strips and Fans

You can use triangle strips and triangle fans to specify an entire surface without having to provide all three vertices for each of the triangles. For example, only seven vertices are required 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.

A triangle fan is similar to a triangle strip, except that all of the triangles share one vertex.



The system uses vertices v1, v2, and v3 to draw the first triangle, v3, v4, and v1 to draw the second triangle, v1, v4, and v5 to draw the third triangle, and so on.

You can use the **wFlags** member of the **D3DTRIANGLE** structure to specify the flags that build triangle strips and fans.

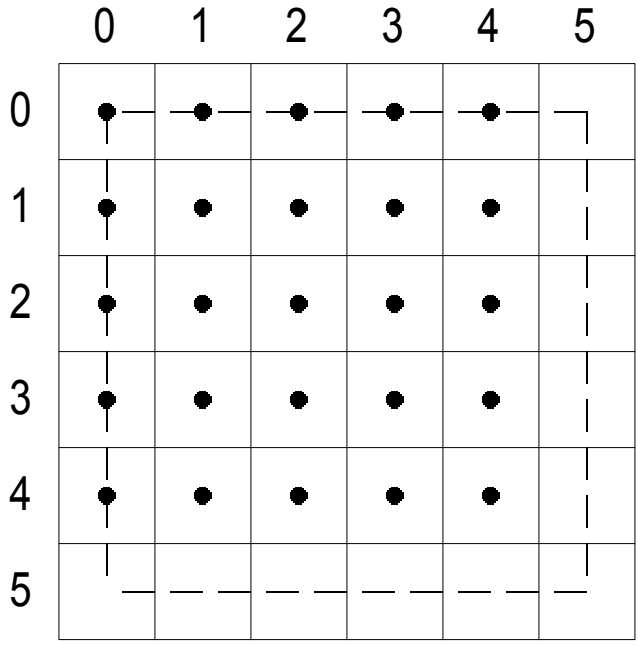
Triangle Rasterization Rules

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.

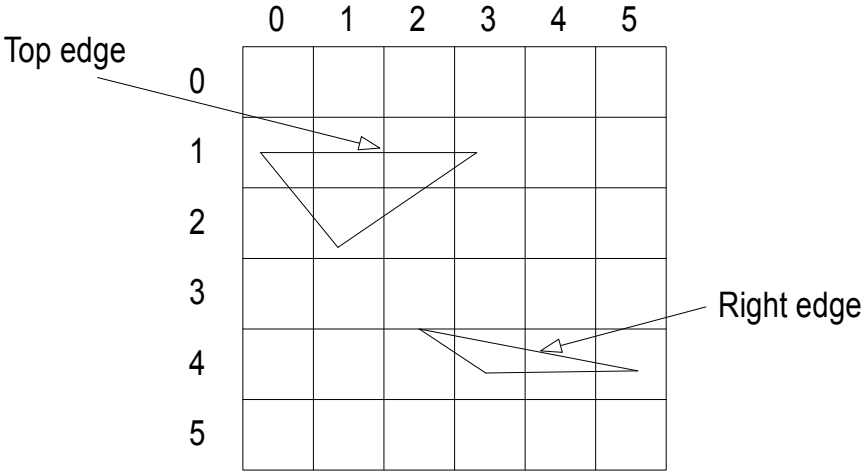
Direct3D uses a top-left filling convention for filling geometry. This is the same convention that is used for rectangles in GDI, the Windows NT polygon rasterizer, and OpenGL. Also, in Direct3D the center of the pixel is the point at which decisions are made; if the center is inside a triangle, the 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. In the future, nearly all manufacturers will implement these rules as they are described in this section.

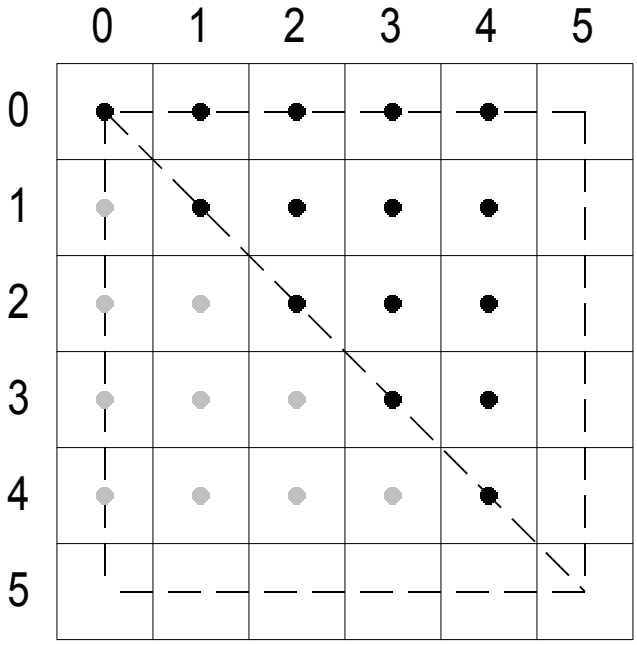
The following illustration shows a rectangle whose upper-left corner is at [0, 0] and whose lower-right corner is at [5, 5]. This rectangle fills 25 pixels, just as you would expect. The width of the rectangle is defined as right–left. The height is defined as bottom–top.



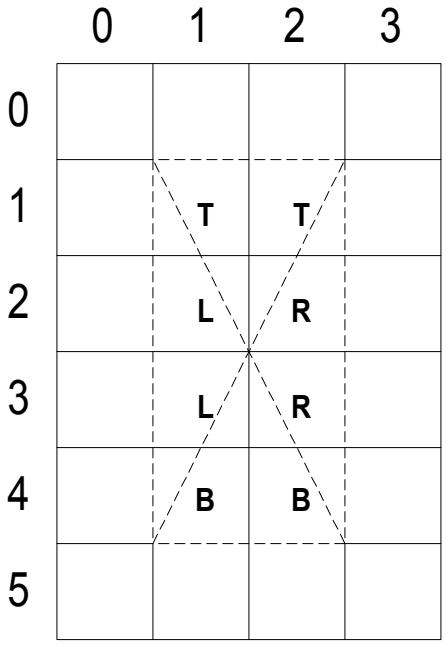
In the top-left filling convention, the word "top" refers to horizontal spans, and the word "left" refers to pixels in spans. An edge cannot be a top edge unless it is horizontal—in the general case, most triangles will have only left and right edges.



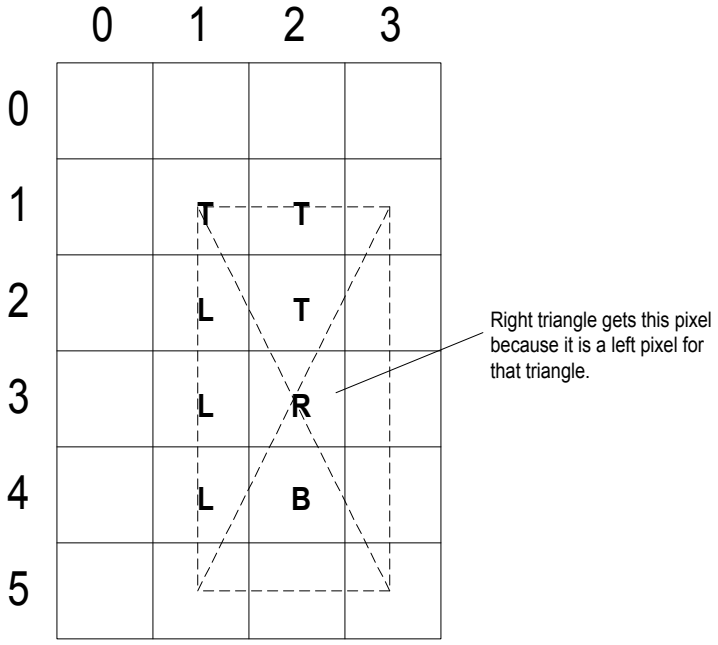
The top-left filling convention determines the action taken by Direct3D when a triangle passes through the center of a pixel. The following illustration shows two triangles, one at [0, 0], [5, 0], and [5, 5], and the other at [0, 5], [0, 0], and [5, 5]. The first triangle in this case gets 15 pixels,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



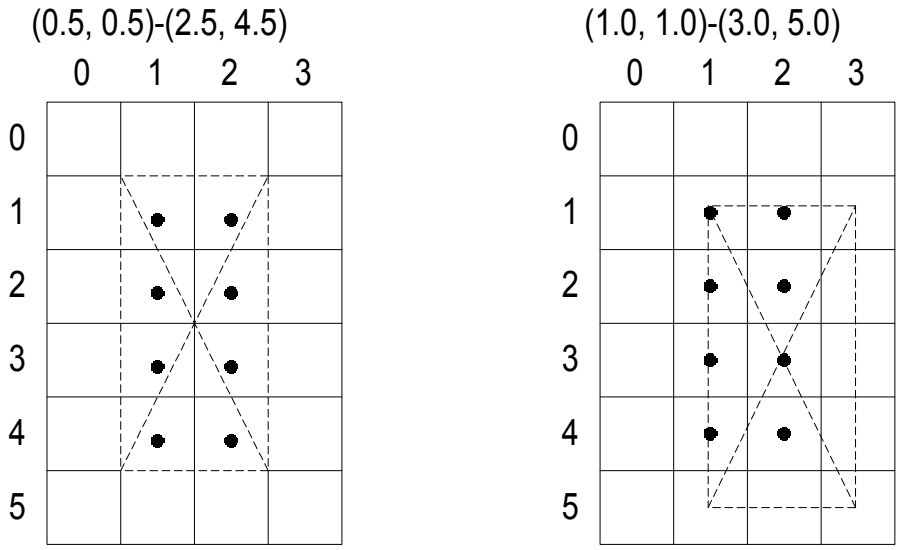
Suppose that a rectangle were defined with its upper-left corner at [0.5, 0.5] and its lower-right corner at [2.5, 4.5]. The center point of this rectangle would be at [1.5, 2.5]. When this rectangle was tessellated, the center of each pixel would be unambiguously inside each of the four triangles, and the top-left filling convention would not be needed.



If a rectangle with the same dimensions were moved slightly, so that its upper-left corner were at [1.0, 1.0], its lower-right corner at [3.0, 5.0], and its center point at [2.0, 3.0], the top-left filling convention would be required. Most of the pixels in this new rectangle would straddle the border between two or more triangles.



Notice that for both rectangles, the same pixels are affected.



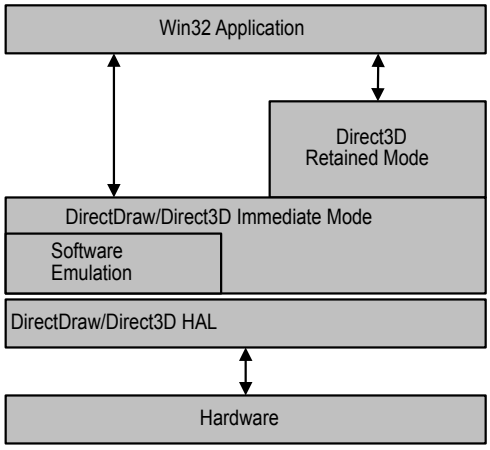
Direct3D Immediate-Mode Architecture

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

- Architectural Overview
- Immediate Mode Object Types
- Immediate Mode COM Interfaces
- The DrawPrimitive Methods and Execute Buffers

Architectural Overview

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 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 COM interface is actually an interface to a DirectDraw object.

Immediate Mode Object Types

Direct3D Immediate Mode is made up of a series of objects. You work with these objects to manipulate your virtual world and build a Direct3D application.

DirectDraw Object

A DirectDraw object provides the functionality of Direct3D; **IDirect3D** and **IDirect3D2** 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, it makes sense that the abilities of Direct3D are incorporated into DirectDraw. You create a DirectDraw object by calling the **DirectDrawCreate** function. For more information, see IDirect3D2 Interface.

DirectDrawSurface Object

A DirectDrawSurface object that was created as a texture map contains the bitmap(s) that your Direct3D application will use as textures. You create an **IDirect3DTexture2** interface by calling the **IDirectDrawSurface3::QueryInterface** method.

For more information, see Textures.

Direct3DDevice Object

A Direct3DDevice object encapsulates and stores the rendering state for an Immediate Mode application; it can be thought of as a rendering target for Direct3D. Prior to DirectX 5, Direct3D devices were interfaces to DirectDrawSurface objects. DirectX 5 introduced a new device-object model, in which a Direct3DDevice object is entirely separate from DirectDraw surfaces. This new object supports the **IDirect3DDevice2** interface.

You can call the **IDirect3D2::CreateDevice** method to create a Direct3DDevice object and retrieve an **IDirect3DDevice2** interface. (Notice that you do not call

QueryInterface to retrieve **IDirect3DDevice2**!) If necessary, you can retrieve an **IDirect3DDevice** interface by calling the **IDirect3DDevice2::QueryInterface** method.

For more information, see The DrawPrimitive Methods and Execute Buffers and Devices.

Direct3DMaterial Object

A Direct3DMaterial object describes the illumination properties of a visible element in a three-dimensional scene, including how it handles light and whether it uses a texture. You can create a Direct3DMaterial object by calling the **IDirect3D2::CreateMaterial** method. You can use the **IDirect3DMaterial2** interface to get and set materials and to retrieve material handles.

For more information, see Materials.

Direct3DViewport Object

A Direct3DViewport object defines the rectangle into which a three-dimensional scene is projected. You can create an **IDirect3DViewport2** interface by calling the **IDirect3D2::CreateViewport** method. For more information, see Viewports and Transformations.

Direct3DLight Object

A Direct3DLight object describes the characteristics of a light in your application. You can use the **IDirect3DLight** interface to get and set lights. You can create an **IDirect3DLight** interface by calling the **IDirect3D2::CreateLight** method. For more information, see Lights.

Direct3DExecuteBuffer Object

A Direct3DExecuteBuffer object is a buffer full of vertices and instructions about how to handle them. Prior to DirectX 5, Immediate-Mode programming was done exclusively by using Direct3DExecuteBuffer objects. The introduction of the DrawPrimitive methods in DirectX 5, however, has made it unnecessary for most applications to work with execute buffers.

For more information about execute buffers, see Execute Buffers.

Immediate Mode COM Interfaces

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

IDirect3D2	Root interface, used to obtain other interfaces
IDirect3DDevice	3D Device for execute-buffer based programming
IDirect3DDevice2	3D Device for DrawPrimitive-based programming
IDirect3DTexture2	Texture-map interface
IDirect3DMaterial2	Surface-material interface
IDirect3DViewport2	Interface to define the screen space viewport's characteristics.
IDirect3DLight	Interface used to work with lights
IDirect3DExecuteBuffer	Interface for working with execute buffers

For backward compatibility with previous versions of DirectX, the following interfaces are also provided. For more information about backward compatibility, see Compatibility with DirectX 3.

IDirect3D

IDirect3DTexture

IDirect3DMaterial

IDirect3DViewport

The DrawPrimitive Methods and Execute Buffers

DirectX 5 introduced a radically new way to use Direct3D Immediate Mode. Previously, you had to fill and execute the execute buffers to accomplish any task. Now, you can use the DrawPrimitive methods, which allow you to draw primitives directly.

The **IDirect3DDevice** interface supports execute buffers. The **IDirect3DDevice2** interface supports the DrawPrimitive methods. Despite the names of these interfaces, **IDirect3DDevice2** is not a COM iteration of the **IDirect3DDevice** interface. Although there is some overlap in the functionality of the interfaces, they are separate implementations. This means that you cannot call **IDirect3DDevice::QueryInterface** to retrieve an **IDirect3DDevice2** interface. You must call the **IDirect3D2::CreateDevice** method, instead.

For more information about the DrawPrimitive methods, see The DrawPrimitive Methods. For more information about working with execute buffers, see Using Execute Buffers. For more information about device objects, see Objects and 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 the COM model. The Immediate Mode API are a very thin layer over the Direct3D drivers.

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

- Immediate-Mode Changes for DirectX 5
- The DrawPrimitive Methods
- GUIDs

- IDirect3D2 Interface
- Devices
- Viewports and Transformations
- Textures
- Lights
- Materials
- Colors and Fog
- Antialiasing
- Direct3D Integration with DirectDraw
- Execute Buffers
- Using Execute Buffers
- States and State Overrides
- Floating-point Precision
- Performance Optimization
- Troubleshooting

Immediate-Mode Changes for DirectX 5

This section discusses some of the important changes in the implementation of Direct3D Immediate Mode for DirectX 5. Information is divided into the following groups.

- Compatibility with DirectX 3
- Moving DirectX 3 Applications to DirectX 5

Compatibility with DirectX 3

Direct3D Immediate Mode has undergone many changes between DirectX 3 and DirectX 5. One of the benefits of the COM model is that it allows additions and modifications to sets of API elements without breaking existing applications. All of the DirectX 3 interfaces are supported in DirectX 5. The DirectX 3 interfaces also work in a similar way (except for a few bug fixes).

The new functionality in DirectX 5 required the addition of many new interfaces. The most important changes in DirectX 5 have been the addition of the DrawPrimitive methods and the new device object model. For more information about the DrawPrimitive methods, see The DrawPrimitive Methods. For more information about the new device object model, see Objects and Interfaces.

Moving DirectX 3 Applications to DirectX 5

Although DirectX 3 applications will work unchanged in DirectX 5, DirectX 3 applications should be modified if the new DirectX 5 features are to be used. The main change required is use the new Direct3D device model described in Objects and Interfaces. Retrieve an **IDirect3D2** interface from DirectDraw instead of **IDirect3D** and then use the **IDirect3D2::CreateDevice** method to create the device object. You can continue to use the **IDirect3DDevice** interface just as you did previously by calling the **IDirect3DDevice2::QueryInterface** method to retrieve the **IDirect3DDevice** interface. The only difference is that if you previously called **IDirect3DDevice::QueryInterface** to retrieve an **IDirectDrawSurface** interface, you should instead call **IDirect3DDevice2::GetRenderTarget**.

The DrawPrimitive Methods

This section discusses the DrawPrimitive methods, an innovation in DirectX 5 that both simplifies Immediate Mode programming and adds new flexibility. Information is divided into the following groups.

- API Extensions for DrawPrimitive
- Architecture of DrawPrimitive Capabilities
- Using Both DrawPrimitive and Execute Buffers
- A Simple DrawPrimitive Example

API Extensions for DrawPrimitive

There is a way to tap into the power of Immediate Mode programming without explicitly using execute buffers. The heart of this system is the **IDirect3DDevice2::DrawPrimitive** method (and its companion, **IDirect3DDevice2::DrawIndexedPrimitive**).

The **IDirect3D** and **IDirect3DDevice** interfaces have been extended to support the ability to draw primitives. These extended versions are called the **IDirect3D2** and **IDirect3DDevice2** interfaces.

To use **IDirect3DDevice2**, retrieve a pointer to the interface by calling the **IDirect3D2::CreateDevice** method. If you need to use some of the methods in **IDirect3DDevice** that are not supported in **IDirect3DDevice2**, you can call **IDirect3DDevice2::QueryInterface** to retrieve a pointer to an **IDirect3DDevice** interface.

The **IDirect3DViewport** interface has also been extended. The new interface, **IDirect3DViewport2**, introduces a closer correspondence between the dimensions of the clipping volume and the viewport than was true for the **IDirect3DViewport** interface.

Architecture of DrawPrimitive Capabilities

The methods provided by the **IDirect3D2** and **IDirect3DDevice2** interfaces enable a user to avoid the execute buffer model. Avoiding the execute buffer model can be useful for two reasons:

1 A new Direct3D developer wants to get up and running quickly.

2 Certain classes of applications (for example, BSP-style games with graphics engines that produce transformed, lit and clipped triangles) do not lend themselves easily to porting to the execute-buffer model. The execute-buffer model does not inherently impose restrictions, but it can be difficult to use.

The **IDirect3DDevice2** interface can be created with the **IDirect3D2::CreateDevice** method. This method takes the DirectDraw surface to render into as a parameter, enabling applications to avoid querying the device interface off of the DirectDraw surface.

A "primitive" in the DrawPrimitive API can be one of the following constructs:

- Point list
- Line list
- Line strip
- Triangle list
- Triangle strip
- Triangle fan

The **IDirect3DDevice2::DrawPrimitive** and **IDirect3DDevice2::DrawIndexedPrimitive** methods draw a primitive in a single call. When possible, these functions call into the driver directly to draw the primitive.

Alternatively, the application can specify the vertices one at a time. To draw a primitive by specifying the vertices individually, the application calls **IDirect3DDevice2::Begin** and specifies the primitive and vertex type, **IDirect3DDevice2::Vertex** to specify each vertex, and **IDirect3DDevice2::End** to finish drawing the primitive. Similarly, to draw an indexed primitive by specifying the indices individually, the application calls **IDirect3DDevice2::BeginIndexed** and specifies the primitive and vertex type, **IDirect3DDevice2::Index** to specify each index, and **IDirect3DDevice2::End** to finish drawing the primitive.

IDirect3DDevice2::Vertex is the only valid method between calls to **IDirect3DDevice2::Begin** and **IDirect3DDevice2::End**.

IDirect3DDevice2::Index is the only valid method between calls to **IDirect3DDevice2::BeginIndexed** and **IDirect3DDevice2::End**.

Note

The DrawPrimitive methods are designed to enable asynchronous operation. Unless you specify D3DDP_WAIT when you call **IDirect3DDevice2::DrawPrimitive** or **IDirect3DDevice2::DrawIndexedPrimitive**, the method will fail if the 3-D hardware cannot currently accept the command, returning

DDERR_WASSTILLDRAWING . This behavior is modeled after the DirectDraw **IDirectDrawSurface3::Blt** operation, where DDBLT_WAIT specifies that the call should return after the command has actually been queued up for execution by the accelerator.

Using Both DrawPrimitive and Execute Buffers

DirectX 5 applications can use both styles of programming in the same application by using the two interfaces to the device object. The application should retrieve an **IDirect3D2** interface by calling the **QueryInterface** method on the DirectDraw object and then use the **IDirect3D2::CreateDevice** method to create a device object. The application should keep the **IDirect3DDevice2** interface thus obtained and also call **IDirect3DDevice2::QueryInterface** to retrieve an **IDirect3DDevice** interface.

It is recommended that you use **IDirect3DMaterial2** and **IDirect3DTexture2** interfaces (created using **IDirect3D2**). You can get handles from these objects using the **IDirect3DDevice2** interface as an argument to the **IDirect3DMaterial2::GetHandle** and **IDirect3DTexture2::GetHandle** methods. You can use these handles both in **IDirect3DDevice2** methods and in execute buffers rendered through **IDirect3DDevice**. This is because **IDirect3DDevice2** and **IDirect3DDevice** are two interfaces to the same underlying object.

Similarly, you can create viewport objects using **IDirect3D2** and use the new **IDirect3DViewport2** interface. These viewport objects can be added to the device using **IDirect3DDevice2::AddViewport**. Because the **IDirect3DViewport2** interface inherits from **IDirect3DViewport**, you can pass it to **IDirect3DDevice** methods that expect the **IDirect3DViewport** interface.

A Simple DrawPrimitive Example

For a complete working example of an application that uses the DrawPrimitive methods, see the files in the Flip3D directory in the samples that ship with the DirectX 5 SDK.

```
/*
 * Constants
 */
#define NUM_VERTICES 3
#define NUM_TRIANGLES 1
D3DTLVERTEX src_v[NUM_VERTICES];
WORD src_t[NUM_TRIANGLES * 3];
DWORD hTex;
D3DSTATEVALUE hMat;

/*
 * A routine that assumes that the above data is initialized
 */
```

```
BOOL RenderScene(LPDIRECT3DDEVICE2 lpDev, LPDIRECT3DVIEWPORT lpView,
   LPD3DRECT lpExtent)
{
   if (IDirect3DDevice2_BeginScene(lpDev) != D3D_OK)
      return FALSE;
   if (IDirect3DDevice2_SetLightState(lpDev, D3DLIGHTSTATE_MATERIAL,
         hMat) != D3D_OK)
      return FALSE;
   if (IDirect3DDevice2_SetRenderState(lpDev,
         D3DRENDERSTATE_TEXTUREHANDLE, hTex) != D3D_OK)
      return FALSE;
   if (IDirect3DDevice2_DrawIndexedPrimitive(lpDev,
         DPT_TRIANGLELIST, DVT_TLVERTEX,
         (LPVOID)src_v, NUM_VERTICES,(LPWORD)src_t, NUM_TRIANGLES*3)
          != D3D_OK)
      return FALSE;
   if (IDirect3DDevice2_EndScene(lpDev) != D3D_OK)
      return FALSE;

   return TRUE;
}
```

GUIDs

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).

IDirect3D2 Interface

The **IDirect3D2** interface is the starting point for creating other Direct3D Immediate Mode interfaces. It is implemented by the DirectDraw object and can be obtained by calling the **IDirectDraw2::QueryInterface** method.

IDirect3D2 lets you find and enumerate the types of Direct3D devices supported by a particular DirectDraw object. It also has methods to create other Direct3D Immediate Mode objects, such as viewports, materials and lights.

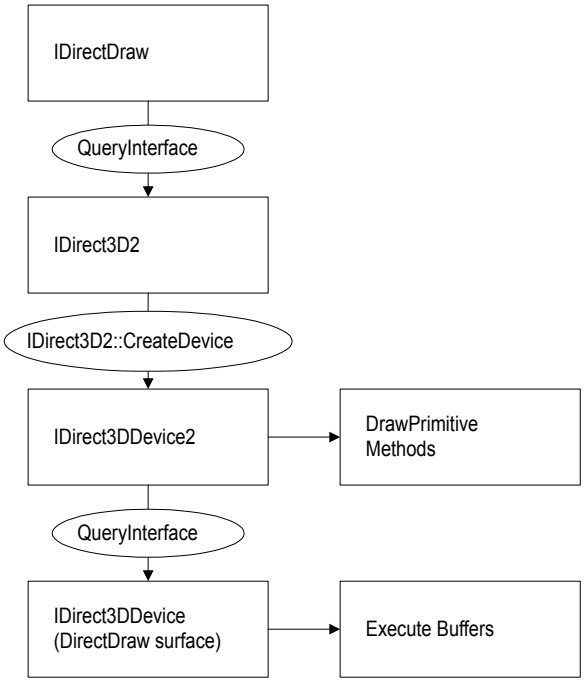
The most important difference between **IDirect3D2** and its predecessor, **IDirect3D**, is that **IDirect3D2** implements an **IDirect3D2::CreateDevice** method. This method creates a Direct3D device that supports the DrawPrimitive methods. For more information about the devices created by the **CreateDevice**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

Devices

A Direct3D device can be thought of as a rendering target for Direct3D. It encapsulates and stores the rendering state.

The Direct3D Immediate Mode device object supports two interfaces, **IDirect3DDevice** and **IDirect3DDevice2**. These two interfaces abstract two styles of programming in Direct3D Immediate Mode. **IDirect3DDevice** allows programming using execute buffers. **IDirect3DDevice2** is a more immediate interface that allows you to draw primitives directly. The interfaces share a few common methods that are useful in either programming style. These have been replicated in both the interfaces to reduce the need to call the **QueryInterface** method between each interface.

You can call the **IDirect3D2::CreateDevice** method to create a Direct3D device object. This method retrieves an **IDirect3DDevice2** interface. The object, however supports both device interfaces; you can retrieve an **IDirect3DDevice** interface by calling the **IDirect3DDevice2::QueryInterface** method.



When you create an **IDirect3DDevice2** interface, the Direct3D device object is a separate object from a DirectDraw surface object. The device object uses a DirectDraw surface as a rendering target. This behavior is different from

IDirect3DDevice, in which Direct3D devices are simply interfaces to DirectDraw surfaces. Keeping devices as separate objects with independent lifetimes from DirectDraw surfaces allows a single 3-D device object to use different DirectDraw surfaces as render targets at different times (for information about this, see **IDirect3DDevice2::SetRenderTarget**). For more information on backward compatibility and the differences between DirectX 3 and DirectX 5, see Compatibility with DirectX 3.

Viewports and Transformations

Direct3D uses three transformations: the view transform, the world transform, and the projection transform. Understanding how these are applied is critical to getting the results that you expect.

This section contains the following topics related to viewports and transformations:

- The Transformation Pipeline
- Setting Transformations
- Creating and Deleting Viewports
- Matrices
- World Transform
- View Transform
- Projection Transform

The Transformation Pipeline

Models are normally created centered around a natural local origin. For instance, it makes sense to have the origin of a chair model be at floor level and centered under the chair. This helps make it easier to place the model in the world. The coordinates that define the model are relative to the origin of the chair model, of course, and are known as model coordinates.

The world transform controls how geometry is transformed from model coordinates into world coordinates. This transform can include translations, rotations, and scalings. You would use the world transform to place your chair model in a room and scale it with respect to the other objects in the room. The world transform applies only to geometry — it does not apply to lights. For an example of working with world transforms, see World Transform.

The view transform controls the transition from world coordinates into "camera space." You can think about this transformation as controlling where the camera appears to be in the world. For an example of working with view transforms, see View Transform.

The projection transform changes the geometry from camera space into "clip space" and applies the perspective distortion. The term "clip space" refers to how the

geometry is clipped to the view volume during this transform. For an example of working with projection transforms, see Projection Transform.

Finally, the geometry in clip space is transformed into pixel coordinates (screen space). This final transformation is controlled by the viewport settings.

Clipping and transforming vertices must take place in homogenous space (simply put, space in which the coordinate system includes a fourth element), but the final result for most applications needs to be non-homogenous 3-D coordinates defined in "screen space." This means that both the input vertices and the clipping volume must be translated into homogenous space to perform the clipping and then translated back into non-homogenous space to be displayed.

The world, view and projection matrices are multiplied in that order to produce the combined transformation matrix [M]. An input vertex [x y z] is considered to be a homogenous vertex [x y z 1]. This vertex is multiplied by the combined 4×4 transform matrix [M] to obtain the output vertex [x1 y1 z1 w]. Following this multiplication, all input vertices are in "post-perspective homogenous space." Now that the vertices have been transformed and changed into homogenous space, the same thing must happen to the clipping volume; it is transformed into the post-perspective homogenous space and the clipping is performed. These clipped vertices (all of which lie within the clip volume) are now transformed back into post-perspective non-homogenous space. As a final step, the points are scaled so that the clip volume maps to the screen space viewport specified by the **dwX**, **dwY**, **dwHeight**, **dwWidth** members of the **D3DVIEWPORT2** structure.

Setting Transformations

Transformations are represented by a 4×4 matrix and are applied using the **IDirect3DDevice2::SetTransform** method. For example, you could use code like this to set the view transform:

```
D3DMATRIX  view;

// fill in the view matrix…

if ((err=lpDev->SetTransform(D3DTRANSFORMSTATE_VIEW, &view)) != D3D_OK)
return err;
```

The render states for setting the projection and the world transformations are D3DTRANSFORMSTATE_PROJECTION and D3DTRANSFORMSTATE_WORLD, respectively.

Creating and Deleting Viewports

A viewport is the surface rectangle into which a three-dimensional scene is projected. A Direct3D viewport object is used to specify the following:

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

- The post-transform clip volume, the contents of which will be mapped in to the viewport. (This is also known as the "window" in a "window-to-viewport transform," in standard computer-graphics terminology.)
- The background material and texture to which the viewport should be cleared.
- The background depth buffer used to initialize the z-buffer before rendering the scene.

The **IDirect3DViewport2** interface has two ways of specifying a viewport. The **D3DVIEWPORT2** structure is specified by the new methods in **IDirect3DViewport2**. This is similar to the **D3DVIEWPORT** structure except that it allows a better clip-volume definition. The new viewport structure is recommended for all DirectX 5 applications.

The first thing to do when creating a viewport is to create a **D3DVIEWPORT2** structure and a pointer to a viewport object:

```
LPDIRECT3DVIEWPORT2  lpD3DViewport;
D3DVIEWPORT2    viewData;
```

Next, fill in the viewport structure:

```
float  aspect = (float)width/height;    // aspect ratio of surface

memset(&viewData, 0, sizeof(D3DVIEWPORT2));
viewData.dwSize = sizeof(D3DVIEWPORT2);
viewData.dwX = 0;
viewData.dwY = 0;
viewData.dwWidth = width;
viewData.dwHeight = height;
viewData.dvClipX = -1.0f;
viewData.dvClipY = aspect;
viewData.dvClipWidth = 2.0f;
viewData.dvClipHeight = 2.0f * aspect;
viewData.dvMinZ = 0.0f;
viewData.dvMaxZ = 1.0f;
```

(You can find a discussion of setting the clipping volume in this structure in the reference material for **D3DVIEWPORT2**.)

After filling in this structure, call the **IDirect3D2::CreateViewport** method to create the viewport object. For this you need a valid **LPDIRECT3D2** pointer, shown in the following example as *lpD3D2*.

```
if ((err = lpD3D2->CreateViewport(&lpD3DViewport2, NULL)) != D3D_OK) {
  return err;
}
```

Now you can call the **IDirect3DDevice2::AddViewport** method to add the newly created viewport object to the device. For this you need a valid **LPDIRECT3DDEVICE2** pointer, shown in the following example as *lpD3DDevice2*.

```
if ((err = d3dapp->lpD3DDevice2->AddViewport(lpD3DViewport2)) != D3D_OK) {
  return err;
}
```

Finally, call the **IDirect3DViewport2::SetViewport2** method to associate the **D3DVIEWPORT2** structure whose values you have already filled out with the new viewport object.

```
if ((err = lpD3DViewport2->SetViewport2(&viewData)) != D3D_OK) {
return err;
}
```

At this point you have a working viewport. If you need to make changes to the viewport values, simply update the values in the **D3DVIEWPORT2** structure and call **IDirect3DViewport2::SetViewport2** again.

When you are ready to delete the viewport, first delete any lights and materials associated with it and then call the **IDirect3DViewport2::Release** method.

```
lpD3DViewport2->Release();
```

Matrices

A Direct3D matrix is a 4×4 homogenous matrix, as defined by a **D3DMATRIX** structure. You use Direct3D matrices to define world, view, and projection transformations. Although Direct3D matrices are not standard objects — they are not represented by a COM interface — you can create and set them just as you would any other Direct3D object.

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 simply 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, you must define D3D_OVERLOADS before including D3dtypes.h.

You can create a Direct3D matrix by calling the **IDirect3DDevice::CreateMatrix** method, and you can set the contents of the matrix by calling the **IDirect3DDevice::SetMatrix** method.

Matrices appear to you only as handles. These handles (defined by the **D3DMATRIXHANDLE** type) are used in execute buffers and in the **D3DOP_MATRIXLOAD** and **D3DOP_MATRIXMULTIPLY** opcodes.

World Transform

The world transform changes coordinates from model space to world space. This can include any combination of translations, rotations, and scalings. For a discussion of the mathematics of transformations, see 3-D Transformations.

You can create a translation using code like this. Notice that here (and in the other transformation samples) the D3D_OVERLOADS form of **D3DMATRIX** is being used.

```
D3DMATRIX Translate(const float dx, const float dy, const float dz)
{
   D3DMATRIX ret = IdentityMatrix();
   ret(3, 0) = dx;
   ret(3, 1) = dy;
   ret(3, 2) = dz;
   return ret;
}   // end of Translate()
```

You can create a rotation around an axis using code like this:

```
D3DMATRIX RotateX(const float rads)
{
   float    cosine, sine;

   cosine = cos(rads);
   sine = sin(rads);
   D3DMATRIX ret = IdentityMatrix();
   ret(1,1) = cosine;
   ret(2,2) = cosine;
   ret(1,2) = -sine;
   ret(2,1) = sine;
   return ret;
}   // end of RotateX()

D3DMATRIX RotateY(const float rads)
{
   float    cosine, sine;

   cosine = cos(rads);
   sine = sin(rads);
   D3DMATRIX ret = IdentityMatrix();
   ret(0,0) = cosine;
   ret(2,2) = cosine;
   ret(0,2) = sine;
   ret(2,0) = -sine;
   return ret;
```

```
}   // end of RotateY()

D3DMATRIX RotateZ(const float rads)
{
    float    cosine, sine;

    cosine = cos(rads);
    sine = sin(rads);
    D3DMATRIX ret = IdentityMatrix();
    ret(0,0) = cosine;
    ret(1,1) = cosine;
    ret(0,1) = -sine;
    ret(1,0) = sine;
    return ret;
}   // end of RotateZ()
```

You can create a scale transform using code like this:

```
D3DMATRIX Scale(const float size)
{
D3DMATRIX ret = IdentityMatrix();
    ret(0, 0) = size;
    ret(1, 1) = size;
    ret(2, 2) = size;
    return ret;
}   // end of Scale()
```

These basic transformations can be combined to create the final transform. Remember that when you combine them the results are not commutative — the order in which you multiply matrices is important.

View Transform

The view transform changes coordinates from world space to camera space. In effect, this transform moves the world around so that the right parts of it are in front of the camera, given a camera position and orientation.

The following *ViewMatrix* function creates a view matrix based on the camera location passed to it. It uses the **Normalize**, **CrossProduct**, and **DotProduct** D3D_OVERLOADS helper functions. The *RotateZ* function it uses is shown in the World Transform section.

```
D3DMATRIX
ViewMatrix(const D3DVECTOR from,      // camera location
        const D3DVECTOR at,        // camera look-at target
        const D3DVECTOR world_up,  // world's up, usually 0, 1, 0
        const float roll)          // clockwise roll around
```

```
                        //    viewing direction,
                        //    in radians
{
   D3DMATRIX  view = IdentityMatrix();  // shown below
   D3DVECTOR  up, right, view_dir;

   view_dir = Normalize(at - from);
   right = CrossProduct(world_up, view_dir);
   up = CrossProduct(view_dir, right);

   right = Normalize(right);
   up = Normalize(up);

   view(0, 0) = right.x;
   view(1, 0) = right.y;
   view(2, 0) = right.z;
   view(0, 1) = up.x;
   view(1, 1) = up.y;
   view(2, 1) = up.z;
   view(0, 2) = view_dir.x;
   view(1, 2) = view_dir.y;
   view(2, 2) = view_dir.z;

   view(3, 0) = -DotProduct(right, from);
   view(3, 1) = -DotProduct(up, from);
   view(3, 2) = -DotProduct(view_dir, from);

   if (roll != 0.0f) {
      // MatrixMult function shown below
      view = MatrixMult(RotateZ(-roll), view);
   }

   return view;
   }  // end of ViewMatrix()

D3DMATRIX
IdentityMatrix(void)       // initializes identity matrix
{
   D3DMATRIX ret;
   for (int i=0; i<4; i++)
      for (int j=0; j<4; j++)
         ret(i, j) = (i==j) ? 1.0f : 0.0f;
   return ret;
}    // end of IdentityMatrix()

// Multiplies two matrices.
```

```
D3DMATRIX
MatrixMult(const D3DMATRIX a, const D3DMATRIX b)
{
   D3DMATRIX ret = ZeroMatrix(); // shown below

   for (int i=0; i<4; i++) {
      for (int j=0; j<4; j++) {
         for (int k=0; k<4; k++) {
            ret(i, j) += a(k, j) * b(i, k);
         }
      }
   }
   return ret;
}   // end of MatrixMult()

D3DMATRIX
ZeroMatrix(void)  // initializes matrix to zero
{
   D3DMATRIX ret;
   for (int i=0; i<4; i++)
      for (int j=0; j<4; j++)
         ret(i, j) = 0.0f;
   return ret;
}   // end of ZeroMatrix()
```

Projection Transform

You can think of the projection transform as controlling the camera's internals. The following *ProjectionMatrix* function takes three input parameters that set the near and far clipping planes and the field of view angle. The field of view should be less than pi.

```
D3DMATRIX
ProjectionMatrix(const float near_plane,     // distance to near clipping plane
                 const float far_plane,      // distance to far clipping plane
                 const float fov)            // field of view angle, in radians
{
   float    c, s, Q;

   c = (float)cos(fov*0.5);
   s = (float)sin(fov*0.5);
   Q = s/(1.0f - near_plane/far_plane);

   D3DMATRIX ret = ZeroMatrix();
   ret(0, 0) = c;
   ret(1, 1) = c;
```

```
    ret(2, 2) = Q;
    ret(3, 2) = -Q*near_plane;
    ret(2, 3) = s;
    return ret;
}   // end of ProjectionMatrix()
```

The following matrix is the projection matrix used by Direct3D. In this formula, *h* is the half-height of the viewing frustum, *F* is the position in z-coordinates of the back clipping plane, and *D* is the position in z-coordinates of the front clipping plane:

$$P = \begin{bmatrix} D/hF & 0 & 0 & 0 \\ 0 & D/hF & 0 & 0 \\ 0 & 0 & F/(F-D) & 1 \\ 0 & 0 & (-F*D)/(F-D) & 0 \end{bmatrix}$$

In Direct3D, the 3,4 element of the projection matrix (the numeral 1 here) cannot be a negative number.

Textures

A texture is a rectangular array of colored pixels. You can think of it as a source of *texels* (texture elements) for the rasterizer. Although the rectangular texture does not necessarily have to be square, the system deals most efficiently with square textures.

You can use textures for texture-mapping faces, in which case their dimensions must be powers of two. If your application uses the RGB color model, you can use 8-, 16-, 24-, and 32-bit textures. If you use the monochromatic (or ramp) color model, however, you can use only 8-bit textures.

This section describes Direct3D textures and the ways your applications can use them.

- Surfaces, Devices, and Handles
- Texture Wrapping
- Texture Filtering and Blending
- Mipmaps
- Transparency and Translucency

Surfaces, Devices, and Handles

You can use a DirectDraw surface as a texture map by calling the **IDirectDrawSurface::QueryInterface** method to retrieve an **IDirect3DTexture2**

interface. You can use the **IDirect3DTexture2** interface to load textures, retrieve handles, and track changes to palettes.

IDirect3D2 and its DirectX 3 counterpart, **IDirect3DTexture**, can be associated with a 3-D device. A texture handle identifies this coupling of a texture map with a device. A texture can be associated with more than one device. When you call the **IDirect3DTexture2::GetHandle** method to associate a texture with a device, it is validated to ensure that the device can support the specified type of texture format and dimensions. The **GetHandle** method returns the texture handle if this validation succeeds. Texture handles can then be used as render states and in materials (for ramp mode) in either **IDirect3DDevice2** or **IDirect3DDevice**. Because a handle obtained by associating a texture object with a device object does not depend on which interfaces were used to obtain it, handles obtained using **IDirect3DTexture** or **IDirect3DTexture2** for a given device object can be used interchangeably.

The **IDirect3DTexture2** interface eliminates some unimplemented methods from the **IDirect3DTexture** interface.

The following example demonstrates how to create an **IDirect3DTexture2** interface and then how to load the texture by calling the **IDirect3DTexture2::GetHandle** and **IDirect3DTexture2::Load** methods. Note that the DirectDraw surface you query must have the **DDSCAPS_TEXTURE** capability to support a Direct3D texture.

```
lpDDS->QueryInterface(IID_IDirect3DTexture2,
    lpD3DTexture2);  // Address of a DIRECT3DTEXTURE object
lpD3DTexture2->GetHandle(
    lpD3DDevice,     // Address of a DIRECT3DDEVICE object
    lphTexture);     // Address of a D3DTEXTUREHANDLE
lpD3DTexture2->Load(
    lpD3DTexture);   // Address of a DIRECT3DTEXTURE object
```

To use a texture handle in an execute buffer, use the **D3DRENDERSTATE_TEXTUREHANDLE** render state (part of the **D3DRENDERSTATETYPE** enumerated type).

Texture Wrapping

The texture coordinates of each face define the region in the texture that is mapped onto that particular face. Your application can use a wrap to calculate texture coordinates.

Your application can use the **D3DRENDERSTATE_WRAPU** and **D3DRENDERSTATE_WRAPV** render states (from the **D3DRENDERSTATETYPE** enumerated type) to specify how the rasterizer should interpret texture coordinates. The rasterizer always interpolates the shortest distance between texture coordinates—that is, a line. The path taken by this line, and the valid values for the u- and v-coordinates, varies with the use of the wrapping flags. If either or both flags is set, the line can wrap around the texture edge in the u- or v- direction, as if the texture had a cylindrical or toroidal topology.

- In flat wrapping mode, in which neither of the wrapping flags is set, the plane specified by the u- and v-coordinates is an infinite tiling of the texture. In this case, values greater than 1.0 are valid for u and v. The shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0.5, 0.5).
- If either D3DRENDERSTATE_WRAPU or D3DRENDERSTATE_WRAPV is set, the texture is an infinite cylinder with a circumference of 1.0. Texture coordinates greater than 1.0 are valid only in the dimension that is not wrapped. The shortest distance between texture coordinates varies with the wrapping flag; if D3DRENDERSTATE_WRAPU is set, the shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0, 0.5).
- If both D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV are set, the texture is a torus. Because the system is closed, texture coordinates greater that 1.0 are invalid. The shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0, 0).

Although texture coordinates that are outside the valid range may be truncated to valid values, this behavior is not defined.

Typically, applications set a wrap flag for cylindrical wraps when the intersection of the texture edges does not match the edges of the face, and do not set a wrap flag when more than half of a texture is applied to a single face.

Texture Filtering and Blending

After a texture has been mapped to a surface, the texture elements (*texels*) of the texture rarely correspond to individual pixels in the final image. A pixel in the final image can correspond to a large collection of texels or to a small piece of a single texel. You can use texture filtering to specify how to interpolate texel values to pixels.

You can use the **D3DRENDERSTATE_TEXTUREMAG** and **D3DRENDERSTATE_TEXTUREMIN** render states (from the **D3DRENDERSTATETYPE** enumerated type) to specify the type of texture filtering to use.

The **D3DRENDERSTATE_TEXTUREMAPBLEND** render state allows you to specify the type of texture blending. Texture blending combines the colors of the texture with the color of the surface to which the texture is being applied. This can be an effective way to achieve a translucent appearance. Texture blending can produce unexpected colors; the best way to avoid this is to ensure that the color of the material is white. The texture-blending options are specified in the **D3DTEXTUREBLEND** enumerated type.

You can use the **D3DRENDERSTATE_SRCBLEND** and **D3DRENDERSTATE_DESTBLEND** render states to specify how colors in the source and destination are combined. The combination options (called *blend factor*) are specified in the **D3DBLEND** enumerated type.

Mipmaps

A mipmap is a sequence of textures, each of which is a progressively lower resolution, prefiltered representation of the same image. Each prefiltered image, or level, in the mipmap is a power of two smaller than the previous level. A high-resolution level is used for objects that are close to the viewer. Lower-resolution levels are used as the object moves farther away. Mipmapping is a computationally low-cost way of improving the quality of rendered textures.

You can use mipmaps when texture-filtering by specifying the appropriate filter mode in the **D3DTEXTUREFILTER** enumerated type. To find out what kinds of mipmapping support are provided by a device, use the flags specified in the **dwTextureFilterCaps** member of the **D3DPRIMCAPS** structure.

In DirectDraw, mipmaps are represented 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.

To create a surface representing a single level of a mipmap, specify the DDSCAPS_MIPMAP flag in the **DDSURFACEDESC** structure passed to the **IDirectDraw2::CreateSurface** method. Because all mipmaps are also textures, the DDSCAPS_TEXTURE flag must also be specified. It is possible to create each level manually and build the chain by using the **IDirectDrawSurface3::AddAttachedSurface** method. However, you can use the **IDirectDraw2::CreateSurface** method to build an entire mipmap chain in a single operation. In this case, the DDSCAPS_COMPLEX flag is also required.

The following example demonstrates building a chain of five mipmap levels of sizes 256×256, 128×128, 64×64, 32×32, and 16×16:

```
DDSURFACEDESC       ddsd;
LPDIRECTDRAWSURFACE3 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 **IDirectDraw2::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 **IDirectDraw2::CreateSurface** will create the number of levels you specify, with a minimum level size of 1×1.

A chain of mipmap surfaces is traversed by using the **IDirectDrawSurface3::GetAttachedSurface** method and specifying the DDSCAPS_MIPMAP and DDSCAPS_TEXTURE flags in the **DDSCAPS** structure. The following example traverses a mipmap chain from highest to lowest resolutions:

```
LPDIRECTDRAWSURFACE lpDDLevel, lpDDNextLevel;
DDSCAPS ddsCaps;

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))
.
.
.
```

You can also build flipping chains of mipmaps. In this scenario, each mipmap level has an associated chain of back buffer texture surfaces. Each back-buffer texture surface is attached to one level of the mipmap. Only the front buffer in the chain has the DDSCAPS_MIPMAP flag set; the others are simply texture maps (created by using the DDSCAPS_TEXTURE flag). A mipmap level can have two attached texture maps, one with DDSCAPS_MIPMAP set, which is the next level in the mipmap chain, and one with the DDSCAPS_BACKBUFFER flag set, which is the back buffer of the flipping chain. All the surfaces in each flipping chain must be of the same size.

It is not possible to build such a surface arrangement with a single call to the **IDirectDraw2::CreateSurface** method. To construct a flipping mipmap, either build a complex mipmap chain and manually attach back buffers by using the **IDirectDrawSurface3::AddAttachedSurface** method, or create a sequence of

flipping chains and build the mipmap by using **IDirectDrawSurface3::AddAttachedSurface**.

Note

Blit operations apply only to a single level in the mipmap chain. To blit an entire chain of mipmaps, each level must be blitted separately.

The **IDirectDrawSurface3::Flip** method will flip all the levels of a mipmap from the level supplied to the lowest level in the mipmap. A destination surface can also be provided, in which case all levels in the mipmap will flip to the back buffer in their flipping chain. This back buffer matches the supplied override. For example, if the third back buffer in the top-level flipping chain is supplied as the override, all levels in the mipmap will flip to the third back buffer.

The number of levels in a mipmap chain is stored explicitly. When an application obtains the surface description of a mipmap (by calling the **IDirectDrawSurface3::Lock** or **IDirectDrawSurface3::GetSurfaceDesc** method), the **dwMipMapCount** member of the **DDSURFACEDESC** structure will contain 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.

Transparency and Translucency

As already mentioned, one method for achieving the appearance of transparent or translucent textures is by using texture blending. You can also use alpha channels and the **D3DRENDERSTATE_BLENDENABLE** render state (from the **D3DRENDERSTATETYPE** enumerated type).

A more straightforward approach to achieving transparency or translucency is to use the DirectDraw support for *color keys*. Color keys are colors or ranges of colors that can be part of either the source or destination of a blit or overlay operation. You can specify whether these colors should always or never be overwritten.

For more information about DirectDraw support for color keys, see Color Keying in the DirectDraw documentation.

Lights

Surfaces are illuminated in your Direct3D application by the lights you create and position. You can use the **IDirect3DLight** interface to get and set lights. You can create an **IDirect3DLight** interface by calling the **IDirect3D2::CreateLight** method.

Lighting is only one of the variables controlling the final appearance of a visible element in a scene. The properties of a surface (including how it reflects light) are determined by materials, which are discussed in Materials. The shading of a surface defines how color is interpreted across a triangle; this is determined by the D3DRENDERSTATE_SHADEMODE render state, in the **D3DRENDERSTATETYPE** enumerated type. Finally, any texture that has been

applied to a visible element also interacts with the lighting to change the object's appearance.

The simplest light type is an ambient light. An ambient light illuminates everything in the scene, regardless of the orientation, position, and surface characteristics of the objects in the scene. Because an ambient light illuminates a scene with equal strength everywhere, the position and orientation of the frame it is attached to are inconsequential. Multiple ambient light sources are combined within a scene.

The color and intensity of the current ambient light are states of the lighting module you can set. You can change the ambient light by using the **D3DLIGHTSTATE_AMBIENT** member of the **D3DLIGHTSTATETYPE** enumerated type.

Direct3D supports four specialized light types in addition to ambient lights. These are defined by the **D3DLIGHTTYPE** enumerated type. You can specify these light types and their capabilities by calling the **IDirect3DLight::SetLight** method and modifying the values in the **D3DLIGHT2** structure.

Point	A light source that radiates equally in all directions from its origin. Point light sources require the system to calculate a new lighting vector for every facet or normal they illuminate, and so are computationally more expensive than a parallel point light source. They produce a more faithful lighting effect than parallel point light sources, however.
Spotlight	A light source that emits a cone of light. Only objects within the cone are illuminated. The cone produces light of two degrees of intensity, with a central brightly lit section (the *umbra*) that acts as a point source, and a surrounding dimly lit section (the *penumbra*) that merges with the surrounding deep shadow. You can specify the angles of each of these two sections by modifying members of the **D3DLIGHT2** structure.
Directional	A light source that is attached to a frame but appears to illuminate all objects with equal intensity, as if it were at an infinite distance from the objects. Directional light has orientation but no position. It is commonly used to simulate distant light sources, such as the sun. It is the best choice of light to use for maximum rendering speed.
Parallel point	A light source that illuminates objects with parallel light, but the orientation of the light is taken from the position of the light source. For example, two meshes on either side of a parallel point light source are lit on the side that faces the

position of the source. The parallel point light source offers similar rendering-speed performance to the directional light source.

You use the **D3DCOLORVALUE** structure to specify the color of your lights. For more information, see Colored Lights in the Colors and Fog section.

Your application can use as many lights as the device supports. To find out how many lights a device supports, call the **IDirect3DDevice2::GetCaps** method and examine the **D3DLIGHTINGCAPS** structure.

Lighting is computationally intensive. By being careful about how you set up your lighting, you can achieve significant performance gains in your application. For detailed lighting performance information, see Lighting Tips in the Performance Optimization section.

To use a light, you first need to create a **D3DLIGHT2** structure and a pointer to a light object.

```
D3DLIGHT2        light;       // Structure defining the light
LPDIRECT3DLIGHT  lpD3DLight;  // Object pointer for the light
```

When you have done this, you need to fill in the **D3DLIGHT2** structure. The following example defines a white point light whose range is set to 10.0.

```
memset(&light, 0, sizeof(D3DLIGHT2));    // clear memory

light.dwSize = sizeof(D3DLIGHT2);       // required
light.dltType = D3DLIGHT_POINT;
light.dvPosition.x = 0.0f;              // set position
light.dvPosition.y = 10.0f;
light.dvPosition.z = 0.0f;
light.dcvColor.r = 1.0f;                // set color to white
light.dcvColor.g = 1.0f;
light.dcvColor.b = 1.0f;
light.dvAttenuation0 = 0.0;             // set linear attenuation
light.dvAttenuation1 = 1.0;
light.dvAttenuation2 = 0.0;
lighht.dvRange = 10.0f;                 // set maximum range
light.dwFlags = D3DLIGHT_ACTIVE;        // enable light
```

The next step is to call the **IDirect3D2::CreateLight** method to create the light object. For this you need a valid **LPDIRECT3D2** interface pointer, *lpD3D2*.

```
if ((err = lpD3D2->CreateLight(&lpD3DLight, NULL) != D3D_OK)
return err;
```

When you have created the light object, you use the **D3DLIGHT2** structure you have already filled in to set its properties. Calling the **IDirect3DLight::SetLight** method associates the **D3DLIGHT2** structure with the light object you just created.

```
if ((err = lpD3DLight->SetLight((D3DLIGHT *)&light)) != D3D_OK)
return err;
```

Finally, you should call the **IDirect3DViewport2::AddLight** method to add this light to your current viewport. (Each light source is bound to a single viewport.)

```
if ((err = lpView->AddLight(lpD3DLight)) != D3D_OK)
return err;
```

At this point you have a new light working with the current viewport. If you need to make changes to the light's values, simply update the **D3DLIGHT2** structure and call the **IDirect3DLight::SetLight** method again.

After you are finished working with the light, you should call the **IDirect3DViewport2::DeleteLight** method to remove the light from the viewport, and then call the **IDirect3DLight::Release** method.

```
if (lpView) {
lpView->DeleteLight(lpD3DLight);
}
RELEASE(lpD3DLight);
```

Materials

A material describes the illumination properties of a surface, including how it handles light and whether it uses a texture. You can also use a material to define the background for a viewport. You can create a material object by calling the **IDirect3D2::CreateMaterial** method. You can use the **IDirect3DMaterial2** interface to get and set materials and to retrieve material handles.

For the ramp-mode software device, the material object keeps track of the texture map used in conjunction with the material. This enables the pre-calculation of the material palettes. When you use textures in ramp mode, you must set the **D3DLIGHTSTATE_MATERIAL** member of the **D3DLIGHTSTATETYPE** enumerated type. Once the material properties are set, a material can be associated with a device. As with textures, a material handle identifies this association of a material and a device. A material can be associated with more than one device. You retrieve a material handle by calling the **IDirect3DMaterial2::GetHandle** method.

The current material is a state variable in a device as part of lighting related states. Handles obtained using **IDirect3DMaterial** or **IDirect3DMaterial2** for a given device object can be used interchangeably.

IDirect3DMaterial2 interface eliminates some unimplemented methods from the **IDirect3DMaterial** interface.

You can define the light-handling properties of a material in four ways:

Ambient	Specifies the color of the ambient light as

	reflected by the material.
Diffuse	Specifies the color of the diffuse light as reflected by the material. Diffuse light is light produced by one of the light sources described by the **D3DLIGHTTYPE** enumerated type (that is, any light except ambient light).
Specular	Specifies the color of reflected highlights as produced by the material.
Emissive	Specifies the color of the light that is emitted by the material.

These light-handling properties and other properties of the material, including the texture handle, are described by the **D3DMATERIAL** structure. You can use the **D3DLIGHTSTATE_MATERIAL** member of the **D3DLIGHTSTATETYPE** enumerated type to identify a material.

You can create an **IDirect3DMaterial2** interface by calling the **IDirect3D2::CreateMaterial** method. The following example demonstrates how to create an **IDirect3DMaterial2** interface. Then it demonstrates how to set the material and retrieve its handle by calling the **IDirect3DMaterial2::SetMaterial** and **IDirect3DMaterial2::GetHandle** methods.

```
lpDirect3D2->CreateMaterial(
    lplpDirect3DMaterial2,  // Address of a new material
    pUnkOuter);             // NULL
lpDirect3DMaterial2->SetMaterial(
    lpD3DMat);              // Address of a D3DMATERIAL structure
lpDirect3DMaterial2->GetHandle(
    lpD3DDevice2,           // Address of a DIRECT3DDEVICE object
    lpD3DMat);              // Address of a D3DMATERIAL structure
```

Colors and Fog

Colors in Direct3D are properties of vertices, textures, materials, faces, lights, and, of course, palettes.

This section describes the Direct3D palette and specular color value capabilities.

- Colored Lights
- Palette Entries
- Fog

Colored Lights

The **dcvColor** member of the **D3DLIGHT2** structure specifies a **D3DCOLORVALUE** structure. The colors defined by this structure 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.

When you use the ramp (monochromatic) lighting mode, the ambient light is built into the ramp, so you can't make your scene any darker than the current ambient light level. Also, remember that colored lights in RGB mode are converted into a gray-scale shade in ramp mode; a red light that looks good in RGB mode will be a dim white light in ramp mode.

Palette Entries

You must be sure to attach a DirectDraw palette to the primary DirectDraw surface to avoid unexpected colors in Direct3D applications. The Direct3D sample code in this SDK attaches the palette to the primary surface whenever the window receives a WM_ACTIVATE message. If you need to track the changes that Direct3D makes to the palette of an 8-bit DirectDraw surface, you can call the **IDirectDrawPalette::GetEntries** method.

Your application can use three flags to specify how it will share palette entries with the rest of the system:

D3DPAL_FREE	The renderer may use this entry freely.
D3DPAL_READONLY	The renderer may not set this entry.
D3DPAL_RESERVED	The renderer may not use this entry.

These flags can be specified in the **peFlags** member of the standard Win32 **PALETTEENTRY** structure. Your application can use these flags when using either the RGB or monochromatic (ramp) renderer. Although you could supply a read-only palette to the RGB renderer, you will get better results with the ramp renderer.

Fog

Fog is simply the alpha part of the color specified in the **specular** member of the **D3DTLVERTEX** structure. Another way of thinking about this is that specular color is really RGBF color, where "F" is "fog."

In monochromatic (ramp) lighting mode, fog is implemented through the light states (that is, the **D3DLIGHTSTATETYPE** enumerated type). In the RGB lighting mode, or when you are working with a HAL, you implement fog by using the **D3DRENDERSTATE_FOGTABLESTART** and **D3DRENDERSTATE_FOGTABLEEND** values in the **D3DRENDERSTATETYPE** enumerated type.

There are three fog modes: linear, exponential, and exponential squared. Only the linear fog mode is currently supported.

When you use linear fog, you specify a start and end point for the fog effect. The fog effect begins at the specified starting point and increases linearly until it reaches its maximum density at the specified end point.

The exponential fog modes begin with a barely visible fog effect and increase to the maximum density along an exponential curve. The following is the formula for the exponential fog mode:

$$f = e^{-(density \times z)}$$

In the exponential squared fog mode, the fog effect increases more quickly than in the exponential fog mode. The following is the formula for the exponential squared fog mode:

$$f = e^{-(density \times z)^2}$$

In these formulas, *e* is the base of the natural logarithms; its value is approximately 2.71828. Note that fog can be considered as a measure of visibility—the lower the fog value, the less visible an object is.

For example, if an application used the exponential fog mode and a fog density of 0.5, the fog value at a distance from the camera of 0.8 would be 0.6703, as shown in the following example:

$$f = \frac{1}{2.71828^{(0.5 \times 0.8)}} = \frac{1}{1.4918} = 0.6703$$

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.

Direct3D supports two antialiasing techniques: edge antialiasing and general antialiasing. Which technique is best for your application depends on your requirements for performance and visual fidelity.

- Edge antialiasing
- General antialiasing

Edge Antialiasing

You can apply edge antialiasing to edges in a scene in your application. In edge antialiasing, you specify the edges in your scene that you want the system to antialias, and the system redraws those edges, 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.

After drawing your scene, use the D3DPRASTERCAPS_ANTIALIASEDGES flag in the **D3DPRIMCAPS** structure to find out whether the current hardware supports edge antialiasing. If it does, set the D3DRENDERSTATE_EDGEANTIALIAS flag to TRUE. Now you can redraw the edges in the scene, using **IDirect3DDevice2::DrawPrimitive** and either the **D3DPT_LINESTRIP** or **D3DPT_LINELIST** primitive type.

Redrawing every edge in your scene will work without introducing major artifacts, but it can be computationally expensive. The most important edges to redraw are those between areas of very different color (for example, silhouette edges) or boundaries between very different materials.

When you have finished antialiasing, set D3DRENDERSTATE_EDGEANTIALIAS to FALSE.

General Antialiasing

Direct3D applies general antialiasing whenever each polygon or line is rendered — no separate pass is required.

On some hardware, general antialiasing can be applied only when the application has rendered the polygons sorted from back to front. To find out whether this is true of the current hardware, you can check the D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT flag (which means that the application must sort the polygons) and the D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT flag (which means that the application need not sort the polygons). These flags are part of the **dwRasterCaps** member of the **D3DPRIMCAPS** structure.

After finding out whether or not you need to sort the polygons, set the render state **D3DRENDERSTATE_ANTIALIAS** to D3DANTIALIAS_SORTDEPENDENT or D3DANTIALIAS_SORTINDEPENDENT and draw the scene.

When you no longer need general antialiasing, disable it by setting **D3DRENDERSTATE_ANTIALIAS** to D3DANTIALIAS_NONE.

Direct3D Integration with DirectDraw

This section contains information about the relationship between DirectDraw and Direct3D objects and interfaces, and about the DirectDraw 3-D–surface capabilities. The following topics are discussed:

- Objects and Interfaces
- Texture Maps
- Z-Buffers
- RGBZ Support

Objects and Interfaces

DirectDraw presents programmers with a single, unified object that encapsulates both the DirectDraw and Direct3D states. When you create a DirectDraw object and then use the **IDirectDraw2::QueryInterface** method to obtain an IDirect3D2 interface, the reference count of the DirectDraw object is 2.

The important implication of this is 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 will be preserved.

In DirectX 2 and DirectX 3, a Direct3D device was aggregated off a DirectDraw surface—that is, **IDirect3DDevice** and **IDirectDrawSurface** were two interfaces to the same object. A given Direct3D object supported multiple 3-D device types. The **IDirect3D** interface was used to find or enumerate the device types. The **IDirect3D::EnumDevices** and **IDirect3D::FindDevice** methods identified the various device types by unique interface IDs (IIDs), which were then used to retrieve a Direct3D device interface by calling the **QueryInterface** method on a DirectDraw surface. The lifetimes of the DirectDraw surface and the Direct3D device were identical, since the same object implemented them. This architecture did not allow the programmer to change the rendering target of the Direct3D device

In DirectX 5 there are two models of Direct3D device objects. In the new model, Direct3D devices are separate objects from DirectDraw surface objects. For backward compatibility with DirectX 3 applications, the earlier model, in which Direct3D devices and DirectDraw surfaces were aggregated, is also supported. You cannot use the new DirectX 5 features with the old model. If your application calls the **QueryInterface** method on a DirectDraw surface and retrieves an **IDirect3DDevice**, it is using the old device model. You cannot call the **QueryInterface** method on a device object created in this way to retrieve an **IDirect3DDevice2** interface.

It is recommended that all applications written with the DirectX 5 SDK use the new device object model.

In DirectX 5, the new device model has the Direct3D device as a separate object from a DirectDraw surface. The **IDirect3D2** interface is used to find or enumerate the types of devices supported. However, **IDirect3D2** identifies devices by unique IDs known as class IDs in COM (CLSID). These are used to uniquely identify one of the various classes that implement a given interface. Since there are multiple Direct3D devices with different capabilities (some software based, some hardware based), but each supports the same set of interfaces, a CLSID is used to identify which type of device object we want. The CLSID obtained from **IDirect3D2::FindDevice** or **IDirect3D2::EnumDevices** is then used in a call to the **IDirect3D2::CreateDevice** method to create a device. The device objects created in this fashion support both **IDirect3DDevice** and **IDirect3DDevice2** interfaces. Unlike in DirectX 3, however, you cannot call QueryInterface on these objects to retrieve an **IDirectDrawSurface** interface. Instead, you must use the **IDirect3DDevice2::GetRenderTarget** method.

The **IDirect3DTexture** interface is not an interface to a distinct object type, but instead is another interface to a DirectDrawSurface object. The same rules for reference counts and state lifetimes that apply to **IDirect3D2** interfaces to DirectDraw objects also apply to Direct3D textures.

The DirectDraw HEL supports the creation of texture, mipmap, and z-buffer surfaces. Furthermore, because of the tight integration of DirectDraw and Direct3D, a DirectDraw-enabled system always provides Direct3D support (in software emulation, at least). Therefore, the DirectDraw HEL exports the DDSCAPS_3DDEVICE flag to indicate that a surface can be used for 3-D rendering. DirectDraw drivers for hardware-accelerated 3-D display cards export this capability to indicate the presence of hardware-accelerated 3-D.

Texture Maps

You can allocate texture map surface by specifying the **DDSCAPS_TEXTURE** flag in the **ddsCaps** member of the **DDSURFACEDESC** structure passed to the **IDirectDraw2::CreateSurface** method.

A wide range of texture pixel formats is supported by the HEL. For a list of these formats, see Texture Map Formats.

Z-Buffers

The DirectDraw HEL can create z-buffers for use by Direct3D or other 3-D–rendering software. The HEL supports 16-bit z-buffers. The DirectDraw device driver for a 3-D–accelerated display card can permit the creation of z-buffers in display memory by exporting the DDSCAPS_ZBUFFER flag. It should also specify the z-buffer depths it supports by using the **dwZBufferBitDepths** member of the **DDCAPS** structure.

An application can clear z-buffers by using the **IDirectDrawSurface3::Blt** method. The DDBLT_DEPTHFILL flag indicates that the blit clears z-buffers. If this flag is specified, the **DDBLTFX** structure passed to the **IDirectDrawSurface3::Blt** method should have its **dwFillDepth** member set to the required z-depth. If the DirectDraw device driver for a 3-D–accelerated display card is designed to provide support for z-buffer clearing in hardware, it should export the DDCAPS_BLTDEPTHFILL flag and should handle DDBLT_DEPTHFILL blits. The destination surface of a depth-fill blit must be a z-buffer.

Note

The actual interpretation of a depth value is specific to the 3-D renderer.

RGBZ Support

DirectDraw supports the RGBZ pixel format. In RGBZ, bits that do not store colors store depth information; instead of being stored in a separate z-buffer, depth information is stored with each pixel.

The RGBZ format is particularly useful for applications that rely on emulating 3-D capabilities in software. Complicated 3-D scenes typically use many small triangles. When z-information is kept in a separate buffer, applications must access random memory locations repeatedly for each line of a triangle; once to check the z-buffer, again to write the new color value (if necessary), and again for the same area (in the common case of overlapping triangles). Applications using RGBZ pixels can perform one memory access for an entire line in a triangle, and there is no additional overhead if the hardware can drop the z-information automatically.

DirectDraw supports copying to an RGBZ surface and clearing it, but it does not contain any methods that directly exploit the depth information.

DirectDraw can copy from a source surface to a destination surface only when the surfaces have exactly the same pixel format. DirectDraw does not emulate copying from an RGBZ surface to an RGB surface.

Some hardware can copy from RGBZ surfaces to RGB surfaces. A useful feature in such copy operations is the ability to drop the z-information automatically. Drivers use the **dwSVBCaps2** member of the **DDCAPS** structure to specify that they can perform RGBZ to RGB conversions. (The **dwSVBCaps** member specifies other capabilities of system- to video-memory blits.)

Most implementations of RGBZ pixel formats support dropping the z-information on blits from system-to video memory, not on blits from video-to-video memory. Blits from system- to video-memory can often be performed asynchronously, leaving more processor time for rendering or logic. In addition, access to z-buffers is read-write intensive and therefore usually occurs in system memory. Applications that sometimes run out of space in video memory (for example, applications that use many textures) will find system- to video-memory blits useful.

Applications can use the **dwFillPixel** member of the **DDBLTFX** structure to apply color fills to RGBZ surfaces. (You cannot fill only the color or only the z-portions of an RGBZ surface—you must set both. The **dwFillPixel** member does this for you.)

Execute Buffers

In the past, all programming with Direct3D Immediate Mode was done using execute buffers. Now that the DrawPrimitive methods have been introduced, however, most new Immediate-Mode programs will not use execute buffers or the **IDirect3DExecuteBuffer** interface. For more information about the DrawPrimitive methods, see The DrawPrimitive Methods.

Execute buffers are similar to the display lists you may be familiar with if you have experience with OpenGL programming. Execute buffers contain a vertex list followed by an instruction stream. The instruction stream consists of operation codes, or *opcodes*, and the data that modifies those opcodes. Each execute buffer is bound to a single Direct3D device.

You can create an **IDirect3DExecuteBuffer** interface by calling the **IDirect3DDevice::CreateExecuteBuffer** method.

```
lpD3DDevice->CreateExecuteBuffer(
    lpDesc,      // Address of a DIRECT3DEXECUTEBUFFERDESC structure
    lplpDirect3DExecuteBuffer,  // Address to contain a pointer to the
                        // Direct3DExecuteBuffer object
    pUnkOuter);   // NULL
```

Execute-buffers reside on a device list. You can use the **IDirect3DDevice::CreateExecuteBuffer** method to allocate space for the actual buffer, which may be on the hardware device.

The buffer is filled with two contiguous arrays of vertices and opcodes by using the following calls to the **IDirect3DExecuteBuffer::Lock**, **IDirect3DExecuteBuffer::Unlock**, and **IDirect3DExecuteBuffer::SetExecuteData** methods:

```
lpD3DExBuf->Lock(
    lpDesc);.     // Address of a DIRECT3DEXECUTEBUFFERDESC structure
//  .
//  .  Store contents through the supplied address
//  .
lpD3DExBuf->Unlock();
lpD3DExBuf->SetExecuteData(
    lpData);      // Address of a D3DEXECUTEDATA structure
```

The last call in the preceding example is to the **IDirect3DExecuteBuffer::SetExecuteData** method. This method notifies Direct3D where the two parts of the buffer reside relative to the address that was returned by the call to the **IDirect3DExecuteBuffer::Lock** method.

You can use the **IDirect3DExecuteBuffer** interface to get and set execute data, and to lock, unlock, optimize, and validate the execute buffer.

Using Execute Buffers

As was pointed out earlier, there are two ways to use Immediate Mode: you can use the DrawPrimitive methods or you can work with execute buffers (display lists). Most developers who have never worked with Immediate Mode before will use the DrawPrimitive methods. Developers who already have an investment in code that uses execute buffers will probably continue to work with them. For more information about the DrawPrimitive methods, see **The DrawPrimitive Methods**.

Execute buffers are complex to understand and fill and are difficult to debug. On the other hand, they allow you to maximize performance. Since communicating with the driver is slow, it makes sense to perform the communication in batches—that is, by using execute buffers.

This section of the documentation describes the contents of execute buffers and how to use them.

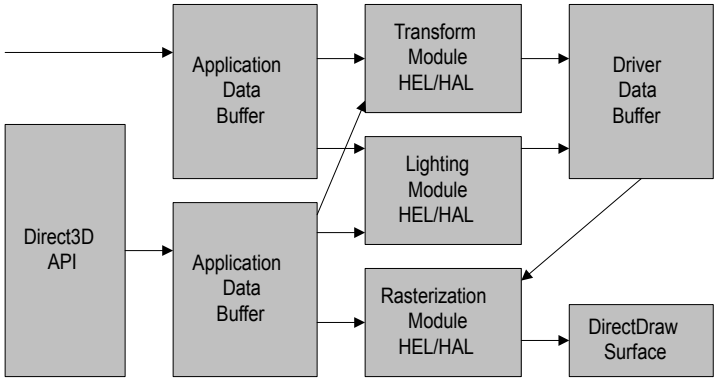
- Execute-Buffer Architecture
- Execute-Buffer Contents
- Creating an Execute Buffer
- Locking the Execute Buffer
- Filling the Execute Buffer
- Unlocking the Execute Buffer
- Executing the Execute Buffer

Execute-Buffer Architecture

Execute buffers are processed first by the transformation module. This module runs through the vertex list, generating transformed vertices by using the state information set up for the transformation module. Clipping can be enabled, generating additional clipping information by using the viewport parameters to clip against. The whole buffer can be rejected here if none of the vertices is visible. Then the vertices are processed by the lighting module, which adds color to them according to the lighting instructions in the execute buffer. Finally, the rasterization module parses the instruction stream, rendering primitives by using the generated vertex information.

When an application calls the **IDirect3DDevice::Execute** method, the system determines whether the vertex list needs to be transformed or transformed and lit. After these operations have been completed, the instruction list is parsed and rendered.

There are really two execute buffers: one for the application and one for the driver. The application data buffer is filled in by the application. It holds geometry (such as vertices and triangles) and state information (the transformation, lighting, and rasterization state) This information persists until the application explicitly changes it. The driver data buffer, on the other hand, holds the output of the transformation and lighting modules (that is, it holds transformed and lit geometry) and hands the data off to the rasterization module. There is only one of these "TL buffers" per driver. The following diagram shows the relationship of these data buffers:



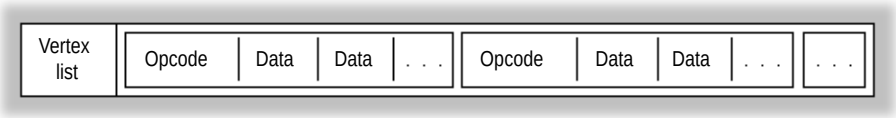
You can disable the lighting module or both the lighting and transformation when you are working with execute buffers. This changes the way the vertex list is interpreted, allowing the user to supply pretransformed or prelit vertices only for the rasterization phase of the rendering pipeline. Note that only one vertex type can be used in each execute buffer. For more information about vertex types, see Vertex Types.

In addition to execute buffers and state changes, Direct3D accepts a third calling mechanism. Either of the transformation or lighting modules can be called directly. This functionality is useful when rasterization is not required, such as when using the transformation module for bounding-box tests.

Execute-Buffer Contents

Execute buffers contain a list of vertices followed by stream of instructions about how to use those vertices. All of these are DWORD-aligned.

The following illustration shows the format of execute buffers.



The instruction stream consists of operation codes, or *opcodes*, and the data that is operated on by those opcodes. The opcodes define how the vertex list should be lit and rendered. Direct3D opcodes are listed in the **D3DOPCODE** enumerated type. The **D3DINSTRUCTION** structure describes instructions in an execute buffer; it contains an opcode, the size of each instruction data unit, and a count of the relevant data units that follow.

One of the most common instructions is a *triangle list* (**D3DOP_TRIANGLE**), which is simply a list of triangle primitives that reference vertices in the vertex list. Because all the primitives in the instruction stream reference vertices in the vertex list only, it is easy for the transformation module to reject a whole buffer of primitives if its vertices are outside the viewing frustum.

Execute Buffer Vertices

Each execute buffer contains a vertex list followed by an instruction stream. The instruction stream defines how the vertex list should be rendered; it is based on indices into the vertex list.

Although you can choose to use transformed and lit vertices (**D3DTLVERTEX**), vertices that have only been lit (**D3DLVERTEX**), or vertices that have been neither transformed nor lit (**D3DVERTEX**), you can have only one of each type of vertex in a single Direct3DExecuteBuffer object. Some execute buffers are used only to change the state of one or more of the modules in the graphics pipeline; these execute buffers do not have vertices.

<table>
<tr><td rowspan="4">Vertices</td><td colspan="2">Vertex 0</td></tr>
<tr><td colspan="2">Vertex 1</td></tr>
<tr><td colspan="2">Vertex 2</td></tr>
<tr><td colspan="2">Vertex 3</td></tr>
<tr><td rowspan="6">Instructions</td><td>0</td><td>Process Vertices</td></tr>
<tr><td rowspan="2">1</td><td>State 0</td></tr>
<tr><td>State 1</td></tr>
<tr><td rowspan="2">2</td><td>Triangle 0</td></tr>
<tr><td>Triangle 1</td></tr>
<tr><td>3</td><td>Exit</td></tr>
</table>

For more information about the handling of vertices in execute buffers, see Vertex Types.

Execute Buffer Instructions

The vertex data in an execute buffer is followed by an instruction stream.

Each instruction is represented by:

- An instruction header
- Opcode
- Byte size
- Number of times this opcode is to be repeated
- Byte offset to first instruction

Execute buffer instructions are commands to the driver. Each instruction is identified by an operation code (opcode). All execute data is prefixed by an instruction header. Data accompanies each iteration of each opcode.

Each opcode can have multiple arguments, including multiple triangles or multiple state changes.

There are only a few main instruction types:

- **Drawing**
- **State changes**
- **Control flow**
- **Others**

Drawing Instructions

The most important of the drawing instructions defines a triangle. In a triangle, vertices are zero-based indices into the vertex list that begins the execute buffer. For more information about triangles, see Triangles.

Other important drawing instructions include line-drawing instructions (**D3DLINE**) and line-drawing instructions (**D3DPOINT**).

State-change Instructions

The system stores the state of each of the modules in the graphics pipeline until the state is overridden by an instruction in an execute buffer.

Transformation state	World, view and projection matrices
Light state	Surface material, fog, ambient lighting
Render state	Texture, antialiasing, z-buffering, and so on

Flow-control Instructions

The flow-control instructions allow you to branch on an instruction or to jump to a new position in the execute buffer, skipping or repeating instructions as necessary. This means that you can use the flow-control instructions as a kind of programming language.

The last flow-control instruction in an execute buffer must be **D3DOP_EXIT**.

Other Instructions

Some other execute-buffer instructions do not fall neatly into the other categories. These include:

Texturing	Download a texture to the device
Matrices	Download or multiply a matrix
Span, SetState	Advanced control for primitives and rendering states.

Creating an Execute Buffer

The tricky thing about creating an execute buffer is figuring out how much memory to allocate for it. There are two basic strategies for determining the correct size:

- Add the sizes of the vertices, opcodes, and data you will be putting into the buffer.
- Allocate a buffer of an arbitrary size and fill it from both ends, putting the vertices at the beginning and the opcodes at the end. When the buffer is nearly full, execute it and allocate another.

The hardware determines the size of the execute buffer. You can retrieve this size by calling the **IDirect3DDevice2::GetCaps** method and examining the **dwMaxBufferSize** member of the **D3DDEVICEDESC** structure. Typically, 64K is a good size for execute buffers when you are using a software driver, because this size makes the best use of the secondary cache. When your application can take advantage

of hardware acceleration, however, it should use smaller execute buffers to take advantage of the primary cache.

After filling in **D3DEXECUTEBUFFERDESC** structure describing your execute buffer, you can call the **IDirect3DDevice::CreateExecuteBuffer** to create it.

For an example of calculating the size of the execute buffer and creating it, see Creating the Scene, in the Direct3D Execute-Buffer Tutorial.

Locking the Execute Buffer

You must lock execute buffers before you can modify them. This action prevents the driver from modifying the buffer while you are working with it.

To lock a buffer, call the **IDirect3DExecuteBuffer::Lock** method. This method takes a single parameter; a pointer to a **D3DEXECUTEBUFFERDESC** structure which, on return, specifies the actual location of the execute buffer's memory.

When working with execute buffers you need to manage three pointers: the execute buffer's start address (retrieved by **IDirect3DExecuteBuffer::Lock**), the instruction start address, and your current position in the buffer. You will use these three pointers to compute vertex offsets, instruction offsets, and the overall size of the execute buffer. When you have finished filling the execute buffer, you will use these pointers to describe the buffer to the driver; for more information about this, see Unlocking the Execute Buffer.

Filling the Execute Buffer

When you have finished filling your execute buffer, it will contain the vertices describing your model and a series of instructions about how the vertices should be interpreted. The following sections describe filling an execute buffer:

- Vertex Types
- Triangles
- Processing Vertices
- Finishing the Instructions

You can streamline the task of filling execute buffers by taking advantages of the helper macros that ship with the samples in the DirectX SDK. The D3dmacs.h header file in the Misc directory of the samples contains many useful macros that will simplify your work.

For an example of filling an execute buffer, see Filling the Execute Buffer, in the Direct3D Execute-Buffer Tutorial.

Vertex Types

Applications that only need to use part of the graphics pipeline can use specialized vertex types into their execute buffers. The following sections discuss three different vertex types:

- Transformed and Lit Vertex
- Lit Vertex
- Vertex (Model Vertex)

You can use one of the helper macros in D3dmacs.h to help you copy data into the execute buffer. This macro is **VERTEX_DATA**.

Transformed and Lit Vertex

The **D3DTLVERTEX** structure defines a transformed and lit vertex (screen coordinates with color). You should use this vertex type when the transform and lighting is not being done through Direct3D.

This vertex type may be the easiest type to use when porting existing 3-D applications to Direct3D.

Lit Vertex

The **D3DLVERTEX** structure defines an untransformed vertex (model coordinates with color). You should use this vertex type when Direct3D will not be used to provide the lighting of these vertex.

Direct3D transforms these vertices prior to rasterization. They are useful for prelit models and scenes with static light sources.

Vertex (Model Vertex)

The **D3DVERTEX** structure defines an untransformed and unlit vertex (model coordinates). You should use this vertex type when you want Direct3D to perform the transformation and the lighting prior to rasterization. This is the vertex type used by Direct3D Retained Mode. It is the best vertex to use if you wants to maximize potential hardware acceleration.

Triangles

You use the **D3DOP_TRIANGLE** opcode to insert a triangle into an execute buffer. In a triangle, vertices are zero-based indices into the vertex list that begins the execute buffer. Triangles are described by the **D3DTRIANGLE** structure.

Triangles are the only geometry type that can be processed by the rasterization module. The screen coordinates range from (0, 0) for the top left of the device (screen or window) to (*width* – 1, *height* – 1) for the bottom right of the device. The depth values range from zero at the front of the viewing frustum to one at the back. Rasterization is performed so that if two triangles that share two vertices are rendered, no pixel along the line joining the shared vertices is rendered twice. The rasterizer culls back facing triangles by determining the winding order of the three vertices of

the triangle. Only those triangles whose vertices are traversed in a clockwise orientation are rendered.

You should be sure that your triangle data is aligned on QWORD (8-byte) boundaries. The **OP_NOP** helper macro in D3dmacs.h can help you with this alignment task. Note that if you use this macro, you must always bracket it with opening and closing braces.

Processing Vertices

After filling in the vertices in your execute buffer, you typically use the **D3DOP_PROCESSVERTICES** opcode to set the lighting and transformations for the vertices.

The **D3DPROCESSVERTICES** structure describes how the vertices should be processed. The **dwFlags** member of this structure specifies the type of vertex you are using in your execute buffer. If you are using **D3DTLVERTEX** vertices, you should specify **D3DPROCESSVERTICES_COPY** for **dwFlags**. For **D3DLVERTEX**, specify **D3DPROCESSVERTICES_TRANSFORM**. For **D3DVERTEX**, specify **D3DPROCESSVERTICES_TRANSFORMLIGHT**.

Finishing the Instructions

The last opcode in your list of instructions should be **D3DOP_EXIT**. This opcode simply signals that the system can stop processing the data.

Unlocking the Execute Buffer

When you have finished filling the execute buffer, you must unlock it. This alerts the driver that it can work with the buffer. You can unlock the buffer by calling the **IDirect3DExecuteBuffer::Unlock** method.

When the execute buffer has been unlocked, call the **IDirect3DExecuteBuffer::SetExecuteData** method to give the driver some important details about the buffer. This method takes a pointer to a **D3DEXECUTEDATA** structure. Among the information you will provide in this structure are the offsets of the vertices and instructions, which you will have been tracking ever since locking the buffer, as described in Locking the Execute Buffer.

Executing the Execute Buffer

Executing an execute buffer is a simple matter of calling the **IDirect3DDevice::Execute** method with pointers to the execute buffer and to the viewport describing the rendering target. You should always check the return value from this method to verify that it was successful.

The *dwFlags* parameter of **IDirect3DDevice::Execute** specifies whether the vertices you supply should be clipped. You should specify D3DEXECUTE_UNCLIPPED if you are using **D3DTLVERTEX** vertices and D3DEXECUTE_CLIPPED otherwise.

When you have executed an execute buffer, you can delete it. This is done simply by calling the **IDirect3DExecuteBuffer::Release** method. You could also use the **RELEASE** macro in D3dmacs.h, if you prefer.

States and State Overrides

Direct3D interprets the data in execute buffers according to the current state settings. Applications set up these states before instructing the system to render data. The **D3DSTATE** structure contains three enumerated types that expose this architecture: **D3DTRANSFORMSTATETYPE**, which sets the state of the transform module; **D3DLIGHTSTATETYPE**, for the lighting module; and **D3DRENDERSTATETYPE**, for the rasterization module.

Each state includes a Boolean value that is essentially a read-only flag. If this flag is set to TRUE, no further state changes are allowed.

Applications can override the read-only state of a module by using the **D3DSTATE_OVERRIDE** macro. This mechanism allows an application to reuse an execute buffer, changing its behavior by changing the system's state. Direct3D Retained Mode uses state overrides to accomplish some tasks that otherwise would require completely rebuilding an execute buffer. For example, the Retained-Mode API uses state overrides to replace the material of a mesh with the material of a frame.

An application might use the **D3DSTATE_OVERRIDE** macro to lock and unlock the Gouraud shade mode, as shown in the following example. (The shade-mode render state is defined by the **D3DRENDERSTATE_SHADEMODE** member of the **D3DRENDERSTATETYPE** enumerated type.)

```
OP_STATE_RENDER(2, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_SHADEMODE, D3DSHADE_GOURAUD, lpBuffer);
    STATE_DATA(D3DSTATE_OVERRIDE(D3DRENDERSTATE_SHADEMODE), TRUE,
lpBuffer);
```

The OP_STATE_RENDER macro implicitly uses the **D3DOP_STATERENDER** opcode, one of the members of the **D3DOPCODE** enumerated type. **D3DSHADE_GOURAUD** is one of the members of the **D3DSHADEMODE** enumerated type.

After executing the execute buffer, the application could use the **D3DSTATE_OVERRIDE** macro again, to allow the shade mode to be changed:

```
    STATE_DATA(D3DSTATE_OVERRIDE(D3DRENDERSTATE_SHADEMODE), FALSE,
lpBuffer);
```

The **OP_STATE_RENDER** and **STATE_DATA** macros are defined in the D3dmacs.h header file in the Misc directory of the DirectX SDK sample.

Floating-point Precision

Direct3D, like the rest of the DirectX architecture, uses a floating-point precision of 53 bits. If your application needs to change this precision, it must change it back to 53 when the calculations are finished. Otherwise, system components that depend on the default value will stop working.

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:

- PC Hardware Accelerators
- Databases and Culling
- Batching Primitives
- Lighting Tips
- Texture Size
- Software versus Hardware
- Triangle Flags
- Clip Tests on Execution
- General Performance Tips

Direct3D applications can use either the ramp driver (for the monochromatic color model) or the RGB driver. The performance notes in the following sections apply to the ramp driver:

- Ramp Textures
- Copy Texture-blending Mode
- Ramp Performance Tips
- Z-Buffer Performance

PC Hardware Accelerators

You will be disappointed by the performance of most of the current 3-D cards. It is unlikely that they will supply a significantly greater polygon throughput rate than unaccelerated hardware. This situation is changing quickly though, with a combination of market forces, design experience, and increasing hardware expertise.

You should be skeptical of published performance figures. They are typically peak rates, produced in ideal conditions for the renderer, and are not actually reproducible in a real-world application.

In the meantime, even if the performance is not what you might desire, many of the current hardware accelerators are still useful. For example, they typically fill polygons very quickly, particularly big polygons. The resolution of your image, the color depth at which you can run, and some special effects might also improve.

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 lights as possible. If you just need to bring up the overall level of lighting, use the ambient light instead of adding a new light. (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. Use the D3DLIGHT_NO_SPECULAR flag in the **D3DLIGHT2** structure as often as reasonable. 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.

Ramp Textures

Applications that use the ramp driver should be conservative with the number of texture colors they require. Each color used in a monochromatic texture requires its own lookup table during rendering. If your application uses hundreds of colors in a scene during rendering, the system must use hundreds of lookup tables, which do not cache well. Also, try to share palettes between textures whenever possible. Ideally, all of your application's textures will fit into one palette, even when you are using a ramp driver with depths greater than 8-bit color.

Copy Texture-blending Mode

Applications that use the ramp driver can sometimes improve performance by using the **D3DTBLEND_COPY** texture-blending mode from the **D3DTEXTUREBLEND** enumerated type. This mode is an optimization for software rasterization; for applications using a HAL, it is equivalent to the **D3DTBLEND_DECAL** texture-blending mode.

Copy mode is the simplest form of rasterization and hence the fastest. When copy mode rasterization is used, no lighting or shading is performed on the texture. The bytes from the texture are copied directly to the screen and mapped onto polygons using the texture coordinates in each vertex. Hence, when using copy mode, your

application's textures must use the same pixel format as the primary surface. They must also use the same palette as the primary surface.

If your application uses the monochromatic model with 8-bit color and no lighting, performance can improve if you use copy mode. If your application uses 16-bit color, however, copy mode is not quite as fast as using modulated textures; for 16-bit color, textures are twice the size as in the 8-bit case, and the extra burden on the cache makes performance slightly worse than using an 8-bit lit texture. You can use D3dtest.exe to verify system performance in this case.

Copy mode implements only two rasterization options, z-buffering and chromakey transparency. The fastest mode is to simply map the texels to the polygons, with no transparency and no z-buffering. Enabling chromakey transparency accelerates the rasterization of invisible pixels because only the texture read is performed, but visible pixels will incur a slight performance degradation because of the chromakey test.

Enabling z-buffering incurs the largest performance degradation for 8 bit copy mode. When z-buffering is enabled, a 16 bit value has to be read and conditionally written per pixel. Even so, enabling z-buffering for copy mode can be faster than disabling it if the average overdraw goes over two and the scene is rendered in front-to-back polygon order.

If your scene has overdraw of less than 2 (which is very likely) you should not use z-buffering in copy mode. The only exception to this rule is if the scene complexity is very high. For example, if you have more than about 1500 rendered polygons in the scene, the sort overhead begins to get high. In that case, it may be worth considering a z-buffer again.

Direct3D is fastest when all it needs to draw is one long triangle instruction. Render state changes just get in the way of this; the longer the average triangle instruction, the better the triangle throughput. Therefore, peak sorting performance can be achieved when all the textures for a given scene are contained in only one texture map or texture page. Although this adds the restriction that no texture coordinate can be larger than 1.0, it has the performance benefit of completely avoiding texture state changes.

For normal simple scenes use one texture, one material, and sort the triangles. Use z-buffering only when the scene becomes complex.

Software versus Hardware

It isn't always obvious that you should use hardware over software renderers. Software renderers work with small polygons efficiently but are not so adept at working with big ones. They use the monochromatic model rather than the RGB model. Hardware renderers, on the other hand, are good at big polygons but less good at small ones. They use the RGB model instead of the monochromatic model. Even if you use hardware, the transformation and lighting are likely to be in software, so CPU speed is still critical to excellent performance.

Triangle Flags

The **wFlags** member of the **D3DTRIANGLE** structure includes flags that allow the system to reuse vertices when building triangle strips and fans. Effective use of these flags allows some hardware to run much faster than it would otherwise.

Applications can use these flags in two ways as acceleration hints to the driver:

D3DTRIFLAG_STARTFLAT(*len*)

> If the current triangle is culled, the driver can also cull the number of subsequent triangles given by *len* in the strip or fan.

D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN

> The driver needs to reload only one new vertex from the triangle and it can reuse the other two vertices from the last triangle that was rendered.

The best possible performance occurs when an application uses both the D3DTRIFLAG_STARTFLAT flag and the D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags.

Because some drivers might not check the D3DTRIFLAG_STARTFLAT flag, applications must be careful when using it. An application using a driver that doesn't check this flag might not render polygons that should have been rendered.

Applications must use the D3DTRIFLAG_START flag before using the D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags. D3DTRIFLAG_START causes the driver to reload all three vertices. All triangles following the D3DTRIFLAG_START flag can use the D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags indefinitely, providing the triangles share edges.

The debugging version of this SDK validates the D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags.

For more information, see Triangle Strips and Fans.

Clip Tests on Execution

Applications that use execute buffers can use the **IDirect3DDevice::Execute** method to render primitives with or without automatic clipping. Using this method without clipping is always faster than setting the clipping flags because clipping tests during either the transformation or rasterization stages slow the process. If your application does not use automatic clipping, however, it must ensure that all of the rendering data is wholly within the viewing frustum. The best way to ensure this is to use simple bounding volumes for the models and transform these first. You can use the results of this first transformation to decide whether to wholly reject the data because all the data is outside the frustum, whether to use the no-clipping version of the **IDirect3DDevice::Execute** method because all the data is within the frustum, or whether to use the clipping flags because the data is partially within the frustum. In Immediate Mode it is possible to set up this sort of functionality within one execute buffer by using the flags in the **D3DSTATUS** structure and the

D3DOP_BRANCHFORWARD member of the **D3DOPCODE** enumerated type to skip geometry when a bounding volume is outside the frustum. Direct3D Retained Mode automatically uses these features to speed up its use of execute buffers.

Ramp Performance Tips

Applications should use the following techniques to achieve the best possible performance when using the monochromatic (ramp) driver:

- Share the same palette among all textures.
- Keep the number of colors in the palette as low as possible—64 or fewer is best.
- Keep the ramp size in materials at 16 or less.
- Make all materials the same (except the texture handle)—allow the textures to specify the coloring. For example, make all the materials white and keep their specular power the same. Many applications do not need more than two materials in a scene: one with a specular power for shiny objects, and one without for matte objects.
- Keep textures as small as possible.
- Fit multiple small textures into a single texture that is 256×256 pixels.
- Render small triangles by using the Gouraud shade mode, and render large triangles by using the flat shade mode.

Developers who must use more than one palette can optimize their code by using one palette as a master palette and ensuring that the other palettes contain a subset of the colors in the master palette.

Z-Buffer Performance

Applications that use the ramp driver 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.

You can also improve the performance of your application by z-testing primitives; that is, by testing a given list of primitives against the z-buffer. If you render the bounding box of a complex object using z-visibility testing, you can easily discover whether the object is completely hidden. If it is hidden, you can avoid even starting to render the object. For example, imagine that the camera is in a room full of 3-D objects. Adjoining this room is a second room full of 3-D objects. The rooms are connected by an open door. If you render the first room and then draw the doorway to the second room using a z-test polygon, you may discover that the doorway is hidden

by one of the objects in the first room and that you don't need to render anything at all in the second room.

You can use the fill-rate test in the D3dtest.exe application that is provided with this SDK to demonstrate overdraw performance for a given driver. (The fill-rate test draws four tunnels from front to back or back to front, depending on the setting you choose.)

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. Although you can use D3dtest.exe to test the speed of system memory against video memory, it cannot predict the performance of your user's personal computer.

You can run all of the Direct3D samples in system memory by using the **-systemmemory** command-line option. This is also useful when developing code because it allows your application to fail in a way that stops the renderer without stopping your system—DirectDraw does not take the WIN16 lock for system-memory surfaces. (The WIN16 lock serializes access to GDI and USER, shutting down Windows for the interval between calls to the **IDirectDrawSurface3::Lock** and **IDirectDrawSurface3::Unlock** methods, as well as between calls to the **IDirectDrawSurface3::GetDC** and **IDirectDrawSurface3::ReleaseDC** methods.)

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.

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
- Nothing Visible
- Debugging
- Borland Floating-Point Initialization
- Miscellaneous

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 palletized 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.

Nothing Visible

If your application runs but nothing is visible, check for the following common errors:

- Ensure that your triangles are not degenerate.
- Make sure that your index lists are internally consistent—that you don't have entries like 1, 2, 2 (which are silently dropped).
- Ensure that your triangles are not being culled.
- Make sure that your transformations are internally consistent.
- Check the viewport to be sure it will allow your triangles to be seen.
- Check the description of the execute buffer.

Debugging

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

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
}
```

Miscellaneous

The following tips can help you uncover common miscellaneous errors:

- Check the memory type (system or video) for your textures.
- Verify that the current hardware can do texturing.
- Make sure that you can restore any lost surfaces.

- Always specify D3DLIGHTSTATE_MATERIAL, even in RGB mode, because it is always necessary in monochromatic mode.

Direct3D Execute-Buffer Tutorial

To create a Direct3D Immediate-Mode application based on execute buffers, you create DirectDraw and Direct3D objects, set render states, fill execute buffers, and execute those buffers.

This section includes a simple Immediate-Mode application that draws a single, rotating, Gouraud-shaded triangle. The triangle is drawn in a window whose size is fixed. For code clarity, we have chosen not to address a number of issues in this sample. For example, full-screen operation, resizing the window, and texture mapping are not included. Furthermore, we have not included some optimizations when their inclusion would have made the code more obscure. Code comments highlight the places in which we did not implement a common optimization.

- Definitions, Prototypes, and Globals
- Enumerating Direct3D Devices
- Creating Objects and Interfaces
- Creating the Scene
- Filling the Execute Buffer
- Animating the Scene
- Rendering
- Working with Matrices
- Restoring and Redrawing
- Releasing Objects
- Error Checking
- Converting Bit Depths
- Main Window Procedure
- WinMain Function

Definitions, Prototypes, and Globals

This section contains the definitions, function prototypes, global variables, constants, and other structural underpinnings for the Imsample.c code sample.

- Header and Includes
- Constants in Imsample.c
- Macros in ImSample.c
- Global Variables
- Function Prototypes

Header and Includes

```
/****************************************************************************
 *
 * File :       imsample.c
 *
 * Author :     Colin D. C. McCartney
 *
 * Date :       1/7/97
 *
 * Version :    V1.1
 *
 ****************************************************************************/

/****************************************************************************
 *
 * Include files
 *
 ****************************************************************************/

#define  INITGUID
#include <windows.h>
#include <math.h>
#include <assert.h>
#include <ddraw.h>
#include <d3d.h>

#include "resource.h"
```

Constants in Imsample.c

```
// Class name for this application's window class.

#define WINDOW_CLASSNAME      "D3DSample1Class"

// Title for the application's window.

#define WINDOW_TITLE          "D3D Sample 1"

// String to be displayed when the application is paused.

#define PAUSED_STRING         "Paused"
```

```
// Half height of the view window.

#define HALF_HEIGHT          D3DVAL(0.5)

// Front and back clipping planes.

#define FRONT_CLIP           D3DVAL(1.0)
#define BACK_CLIP            D3DVAL(1000.0)

// Fixed window size.

#define WINDOW_WIDTH         320
#define WINDOW_HEIGHT        200

// Maximum length of the chosen device name and description of the
// chosen Direct3D device.

#define MAX_DEVICE_NAME      256
#define MAX_DEVICE_DESC      256

// Amount to rotate per frame.

#define M_PI                 3.14159265359
#define M_2PI                6.28318530718
#define ROTATE_ANGLE_DELTA   (M_2PI / 300.0)

// Execute buffer contents

#define NUM_VERTICES         3
#define NUM_INSTRUCTIONS     6
#define NUM_STATES           7
#define NUM_PROCESSVERTICES  1
#define NUM_TRIANGLES        1
```

Macros in Imsample.c

```
// Extract the error code from an HRESULT

#define CODEFROMHRESULT(hRes) ((hRes) & 0x0000FFFF)

#ifdef _DEBUG
#define ASSERT(x)      assert(x)
#else
#define ASSERT(x)
#endif
```

```
// Used to keep the compiler from issuing warnings about any unused
// parameters.

#define USE_PARAM(x)    (x) = (x)
```

Global Variables

```
// Application instance handle (set in WinMain).

static HINSTANCE            hAppInstance            = NULL;

// Running in debug mode?

static BOOL                 fDebug                  = FALSE;

// Is the application active?

static BOOL                 fActive                 = TRUE;

// Has the application been suspended?

static BOOL                 fSuspended              = FALSE;

// DirectDraw interfaces

static LPDIRECTDRAW         lpdd                    = NULL;
static LPDIRECTDRAWSURFACE  lpddPrimary             = NULL;
static LPDIRECTDRAWSURFACE  lpddDevice              = NULL;
static LPDIRECTDRAWSURFACE  lpddZBuffer             = NULL;
static LPDIRECTDRAWPALETTE  lpddPalette             = NULL;

// Direct3D interfaces

static LPDIRECT3D           lpd3d                   = NULL;
static LPDIRECT3DDEVICE     lpd3dDevice             = NULL;
static LPDIRECT3DMATERIAL   lpd3dMaterial           = NULL;
static LPDIRECT3DMATERIAL   lpd3dBackgroundMaterial = NULL;
static LPDIRECT3DVIEWPORT   lpd3dViewport           = NULL;
static LPDIRECT3DLIGHT      lpd3dLight              = NULL;
static LPDIRECT3DEXECUTEBUFFER lpd3dExecuteBuffer   = NULL;

// Direct3D handles

static D3DMATRIXHANDLE      hd3dWorldMatrix         = 0;
```

```
static D3DMATRIXHANDLE      hd3dViewMatrix          = 0;
static D3DMATRIXHANDLE      hd3dProjMatrix          = 0;
static D3DMATERIALHANDLE    hd3dSurfaceMaterial     = 0;
static D3DMATERIALHANDLE    hd3dBackgroundMaterial  = 0;

// Globals used for selecting the Direct3D device. They are
// globals because this makes it easy for the enumeration callback
// function to read and write from them.

static BOOL                 fDeviceFound            = FALSE;
static DWORD                dwDeviceBitDepth        = 0;
static GUID                 guidDevice;
static char                 szDeviceName[MAX_DEVICE_NAME];
static char                 szDeviceDesc[MAX_DEVICE_DESC];
static D3DDEVICEDESC        d3dHWDeviceDesc;
static D3DDEVICEDESC        d3dSWDeviceDesc;

// The screen coordinates of the client area of the window. This
// rectangle defines the destination into which we blit to update
// the client area of the window with the results of the 3-D rendering.

static RECT                 rDstRect;

// This rectangle defines the portion of the rendering target surface
// into which we render. The top-left coordinates of this rectangle
// are always zero; the right and bottom coordinates give the size of
// the viewport.

static RECT                 rSrcRect;

// Angle of rotation of the world matrix.

static double               dAngleOfRotation        = 0.0;

// Predefined transformations.

static D3DMATRIX d3dWorldMatrix =
{
    D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 1.0)
};

static D3DMATRIX d3dViewMatrix =
{
```

```
    D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 5.0), D3DVAL( 1.0)
};

static D3DMATRIX d3dProjMatrix =
{
    D3DVAL( 2.0), D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 2.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 1.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL(-1.0), D3DVAL( 0.0)
};
```

Function Prototypes

```
static void         ReportError(HWND hwnd, int nMessage,
                        HRESULT hRes);
static void         FatalError(HWND hwnd, int nMessage, HRESULT hRes);

static DWORD        BitDepthToFlags(DWORD dwBitDepth);
static DWORD        FlagsToBitDepth(DWORD dwFlags);

static void         SetPerspectiveProjection(LPD3DMATRIX lpd3dMatrix,
                                double      dHalfHeight,
                                double      dFrontClipping,
                                double      dBackClipping);
static void         SetRotationAboutY(LPD3DMATRIX lpd3dMatrix,
                            double      dAngleOfRotation);

static HRESULT      CreateDirect3D(HWND hwnd);
static HRESULT      ReleaseDirect3D(void);

static HRESULT      CreatePrimary(HWND hwnd);
static HRESULT      RestorePrimary(void);
static HRESULT      ReleasePrimary(void);

static HRESULT WINAPI EnumDeviceCallback(LPGUID      lpGUID,
                            LPSTR           lpszDeviceDesc,
                            LPSTR           lpszDeviceName,
                            LPD3DDEVICEDESC lpd3dHWDeviceDesc,
                            LPD3DDEVICEDESC lpd3dSWDeviceDesc,
                            LPVOID          lpUserArg);
static HRESULT      ChooseDevice(void);
```

```
static HRESULT      CreateDevice(DWORD dwWidth, DWORD dwHeight);
static HRESULT      RestoreDevice(void);
static HRESULT      ReleaseDevice(void);

static LRESULT      RestoreSurfaces(void);

static HRESULT      FillExecuteBuffer(void);
static HRESULT      CreateScene(void);
static HRESULT      ReleaseScene(void);
static HRESULT      AnimateScene(void);

static HRESULT      UpdateViewport(void);

static HRESULT      RenderScene(void);
static HRESULT      DoFrame(void);
static void         PaintSuspended(HWND hwnd, HDC hdc);

static LRESULT      OnMove(HWND hwnd, int x, int y);
static LRESULT      OnSize(HWND hwnd, int w, int h);
static LRESULT      OnPaint(HWND hwnd, HDC hdc, LPPAINTSTRUCT lpps);
static LRESULT      OnIdle(HWND hwnd);

LRESULT CALLBACK    WndProc(HWND hwnd, UINT msg,
                    WPARAM wParam, LPARAM lParam);
int PASCAL          WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszCommandLine, int cmdShow);
```

Enumerating Direct3D Devices

The first thing a Direct3D application should do is enumerate the available Direct3D device drivers. The most important API element in this job is **IDirect3D2::EnumDevices**.

This section contains the ChooseDevice function that selects among the available Direct3D devices and the EnumDeviceCallback function that implements the selection mechanism.

- Enumeration Callback Function
- Enumeration Function

This sample application does not demonstrate the enumeration of display modes, which you will need to do if your application supports full-screen rendering modes. To enumerate the display modes, call the **IDirectDraw2::EnumDisplayModes** method.

Enumeration Callback Function

The EnumDeviceCallback function is invoked for each Direct3D device installed on the system. For each device we retrieve its identifying GUID, a name and description, a description of its hardware and software capabilities, and an unused user argument.

The EnumDeviceCallback function uses the following algorithm to choose an appropriate Direct3D device:

1. Discard any devices which don't match the current display depth.
2. Discard any devices which can't do Gouraud-shaded triangles.
3. If a hardware device is found which matches points one and two, use it. However, if we are running in debug mode we will skip hardware.
4. Otherwise favor Mono/Ramp mode software renderers over RGB ones; until MMX is widespread, Mono will be faster.

This callback function is invoked by the ChooseDevice enumeration function, which is described in Enumeration Function.

Note that the first parameter passed to this callback function, lpGUID, is NULL for the primary device. All other devices should have a non-NULL pointer. You should consider saving the actual GUID for the device you choose, not the pointer to the GUID, in case the pointer is accidentally corrupted.

```
static HRESULT WINAPI
EnumDeviceCallback(LPGUID         lpGUID,
                   LPSTR          lpszDeviceDesc,
                   LPSTR          lpszDeviceName,
                   LPD3DDEVICEDESC lpd3dHWDeviceDesc,
                   LPD3DDEVICEDESC lpd3dSWDeviceDesc,
                   LPVOID         lpUserArg)
{
    BOOL           fIsHardware;
    LPD3DDEVICEDESC lpd3dDeviceDesc;

        // Call the USE_PARAM macro on the unused parameter to
        // avoid compiler warnings.

    USE_PARAM(lpUserArg);

        // If there is no hardware support the color model is zero.

    fIsHardware     = (0 != lpd3dHWDeviceDesc->dcmColorModel);
    lpd3dDeviceDesc = (fIsHardware ? lpd3dHWDeviceDesc :
                                     lpd3dSWDeviceDesc);

        // If we are in debug mode and this is a hardware device,
        // skip it.
```

```
if (fDebug && fIsHardware)
    return D3DENUMRET_OK;

    // Does the device render at the depth we want?

if (0 == (lpd3dDeviceDesc->dwDeviceRenderBitDepth &
        dwDeviceBitDepth))
{
          // If not, skip this device.

    return D3DENUMRET_OK;
}

    // The device must support Gouraud-shaded triangles.

if (D3DCOLOR_MONO == lpd3dDeviceDesc->dcmColorModel)
{
    if (!(lpd3dDeviceDesc->dpcTriCaps.dwShadeCaps &
        D3DPSHADECAPS_COLORGOURAUDMONO))
    {
                // No Gouraud shading. Skip this device.

        return D3DENUMRET_OK;
    }
}
else
{
    if (!(lpd3dDeviceDesc->dpcTriCaps.dwShadeCaps &
        D3DPSHADECAPS_COLORGOURAUDRGB))
    {
                // No Gouraud shading. Skip this device.

        return D3DENUMRET_OK;
    }
}

if (!fIsHardware && fDeviceFound &&
      (D3DCOLOR_RGB == lpd3dDeviceDesc->dcmColorModel))
{
          // If this is software RGB and we already have found
          // a software monochromatic renderer, we are not
          // interested. Skip this device.

    return D3DENUMRET_OK;
}
```

```
        // This is a device we are interested in. Save the details.

    fDeviceFound = TRUE;
    CopyMemory(&guidDevice, lpGUID, sizeof(GUID));
    strcpy(szDeviceDesc, lpszDeviceDesc);
    strcpy(szDeviceName, lpszDeviceName);
    CopyMemory(&d3dHWDeviceDesc, lpd3dHWDeviceDesc,
            sizeof(D3DDEVICEDESC));
    CopyMemory(&d3dSWDeviceDesc, lpd3dSWDeviceDesc,
            sizeof(D3DDEVICEDESC));

        // If this is a hardware device, we have found
        // what we are looking for.

    if (fIsHardware)
        return D3DENUMRET_CANCEL;

        // Otherwise, keep looking.

    return D3DENUMRET_OK;
}
```

Enumeration Function

The ChooseDevice function invokes the EnumDeviceCallback function , which is described in Enumeration Callback Function.

```
static HRESULT
ChooseDevice(void)
{
    DDSURFACEDESC ddsd;
    HRESULT       hRes;

    ASSERT(NULL != lpd3d);
    ASSERT(NULL != lpddPrimary);

        // Since we are running in a window, we will not be changing the
        // screen depth; therefore, the pixel format of the rendering
        // target must match the pixel format of the current primary
        // surface. This means that we need to determine the pixel
        // format of the primary surface.

    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
```

```
    hRes = lpddPrimary->lpVtbl->GetSurfaceDesc(lpddPrimary, &ddsd);
    if (FAILED(hRes))
        return hRes;

    dwDeviceBitDepth =
        BitDepthToFlags(ddsd.ddpfPixelFormat.dwRGBBitCount);

        // Enumerate the devices and pick one.

    fDeviceFound = FALSE;
    hRes = lpd3d->lpVtbl->EnumDevices(lpd3d, EnumDeviceCallback,
                                &fDeviceFound);
    if (FAILED(hRes))
        return hRes;

    if (!fDeviceFound)
    {
            // No suitable device was found. We cannot allow
            // device-creation to continue.

        return DDERR_NOTFOUND;
    }

    return DD_OK;
}
```

Creating Objects and Interfaces

This section contains functions that create the primary DirectDraw surface, a DirectDrawClipper object, a Direct3D object, and a Direct3DDevice.

- Creating the Primary Surface and Clipper Object
- Creating the Direct3D Object
- Creating the Direct3D Device

Creating the Primary Surface and Clipper Object

The CreatePrimary function creates the primary surface (representing the desktop) and creates and attaches a clipper object. If necessary, this function also creates a palette.

```
static HRESULT
CreatePrimary(HWND hwnd)
{
```

```
HRESULT             hRes;
DDSURFACEDESC       ddsd;
LPDIRECTDRAWCLIPPER lpddClipper;
HDC                 hdc;
int                 i;
PALETTEENTRY        peColorTable[256];

ASSERT(NULL != hwnd);
ASSERT(NULL != lpdd);
ASSERT(NULL == lpddPrimary);
ASSERT(NULL == lpddPalette);

// Create the primary surface.

ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize         = sizeof(ddsd);
ddsd.dwFlags        = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
hRes = lpdd->lpVtbl->CreateSurface(lpdd, &ddsd, &lpddPrimary, NULL);
if (FAILED(hRes))
   return hRes;

// Create the clipper. We bind the application's window to the
// clipper and attach it to the primary. This ensures that when we
// blit from the rendering surface to the primary we don't write
// outside the visible region of the window.

hRes = DirectDrawCreateClipper(0, &lpddClipper, NULL);
if (FAILED(hRes))
   return hRes;
hRes = lpddClipper->lpVtbl->SetHWnd(lpddClipper, 0, hwnd);
if (FAILED(hRes))
{
   lpddClipper->lpVtbl->Release(lpddClipper);
   return hRes;
}
hRes = lpddPrimary->lpVtbl->SetClipper(lpddPrimary, lpddClipper);
if (FAILED(hRes))
{
   lpddClipper->lpVtbl->Release(lpddClipper);
   return hRes;
}

// We release the clipper interface after attaching it to the
// surface because we don't need to use it again. The surface
// holds a reference to the clipper when it has been attached.
```

```
// The clipper will therefore be released automatically when the
// surface is released.

lpddClipper->lpVtbl->Release(lpddClipper);

// If the primary surface is palettized, the device will be, too.
// (The device surface must have the same pixel format as the
// current primary surface if we want to double buffer with
// DirectDraw.) Therefore, if the primary surface is palettized we
// need to create a palette and attach it to the primary surface
// (and to the device surface when we create it).

ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
hRes = lpddPrimary->lpVtbl->GetSurfaceDesc(lpddPrimary, &ddsd);
if (FAILED(hRes))
   return hRes;
if (ddsd.ddpfPixelFormat.dwFlags & DDPF_PALETTEINDEXED8)
{
// Initializing the palette correctly is essential. Since we are
// running in a window, we must not change the top ten and bottom
// ten static colors. Therefore, we copy them from the system
// palette and mark them as read only (D3DPAL_READONLY). The middle
// 236 entries are free for use by Direct3D so we mark them free
// (D3DPAL_FREE).

// NOTE: In order that the palette entries are correctly
// allocated it is essential that the free entries are
// also marked reserved to GDI (PC_RESERVED).

// NOTE: We don't need to specify the palette caps flag
// DDPCAPS_INITIALIZE. This flag is obsolete. CreatePalette
// must be given a valid palette-entry array and always
// initializes from it.

   hdc = GetDC(NULL);
   GetSystemPaletteEntries(hdc, 0, 256, peColorTable);
   ReleaseDC(NULL, hdc);

   for (i = 0; i < 10; i++)
      peColorTable[i].peFlags = D3DPAL_READONLY;
   for (i = 10; i < 246; i++)
      peColorTable[i].peFlags = D3DPAL_FREE | PC_RESERVED;
   for (i = 246; i < 256; i++)
      peColorTable[i].peFlags = D3DPAL_READONLY;
   hRes = lpdd->lpVtbl->CreatePalette(lpdd,
```

```
            DDPCAPS_8BIT, peColorTable, &lpddPalette, NULL);

        if (FAILED(hRes))
            return hRes;

        hRes = lpddPrimary->lpVtbl->SetPalette(lpddPrimary,
                lpddPalette);
            return hRes;
    }

    return DD_OK;
}
```

Creating the Direct3D Object

The CreateDirect3D function creates the DirectDraw (Direct3D) driver objects and retrieves the COM interfaces for communicating with these objects. This function calls three crucial API elements: **DirectDrawCreate**, to create the DirectDraw object, **IDirectDraw2::SetCooperativeLevel**, to determine whether the application will run in full-screen or windowed mode, and **IDirectDraw::QueryInterface**, to retrieve a pointer to the Direct3D interface.

```
static HRESULT
CreateDirect3D(HWND hwnd)
{
    HRESULT hRes;

    ASSERT(NULL == lpdd);
    ASSERT(NULL == lpd3d);

    // Create the DirectDraw/3D driver object and get the DirectDraw
    // interface to that object.

    hRes = DirectDrawCreate(NULL, &lpdd, NULL);
    if (FAILED(hRes))
        return hRes;

    // Since we are running in a window, set the cooperative level to
    // normal. Also, to ensure that the palette is realized correctly,
    // we need to pass the window handle of the main window.

    hRes = lpdd->lpVtbl->SetCooperativeLevel(lpdd, hwnd, DDSCL_NORMAL);
    if (FAILED(hRes))
        return hRes;

        // Retrieve the Direct3D interface to the DirectDraw/3D driver
```

```
    // object.

  hRes = lpdd->lpVtbl->QueryInterface(lpdd, &IID_IDirect3D, &lpd3d);
  if (FAILED(hRes))
    return hRes;

  return DD_OK;
}
```

Creating the Direct3D Device

The CreateDevice function creates an instance of the Direct3D device we chose earlier, using the specified width and height.

This function handles all aspects of the device creation, including choosing the surface-memory type, creating the device surface, creating the z-buffer (if necessary), and attaching the palette (if required). If you create a z-buffer, you must do so before creating an **IDirect3DDevice** interface.

```
static HRESULT
CreateDevice(DWORD dwWidth, DWORD dwHeight)
{
  LPD3DDEVICEDESC lpd3dDeviceDesc;
  DWORD           dwDeviceMemType;
  DWORD           dwZBufferMemType;
  DDSURFACEDESC   ddsd;
  HRESULT         hRes;
  DWORD           dwZBufferBitDepth;

  ASSERT(NULL != lpdd);
  ASSERT(NULL != lpd3d);
  ASSERT(NULL != lpddPrimary);
  ASSERT(NULL == lpddDevice);
  ASSERT(NULL == lpd3dDevice);

  // Determine the kind of memory (system or video) from which the
  // device surface should be allocated.

  if (0 != d3dHWDeviceDesc.dcmColorModel)
  {
    lpd3dDeviceDesc = &d3dHWDeviceDesc;

    // Device has a hardware rasterizer. Currently this means that
    // the device surface must be in video memory.

    dwDeviceMemType  = DDSCAPS_VIDEOMEMORY;
```

```
    dwZBufferMemType = DDSCAPS_VIDEOMEMORY;
}
else
{
    lpd3dDeviceDesc = &d3dSWDeviceDesc;

    // Device has a software rasterizer. We will let DirectDraw
    // decide where the device surface resides unless we are
    // running in debug mode, in which case we will force it into
    // system memory. For a software rasterizer the z-buffer should
    // always go into system memory. A z-buffer in video memory will
    // seriously degrade the application's performance.

    dwDeviceMemType  = (fDebug ? DDSCAPS_SYSTEMMEMORY : 0);
    dwZBufferMemType = DDSCAPS_SYSTEMMEMORY;
}

// Create the device surface. The pixel format will be identical
// to that of the primary surface, so we don't have to explicitly
// specify it. We do need to explicitly specify the size, memory
// type and capabilities  of the surface.

ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize         = sizeof(ddsd);
ddsd.dwFlags        = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;
ddsd.dwWidth        = dwWidth;
ddsd.dwHeight       = dwHeight;
ddsd.ddsCaps.dwCaps = DDSCAPS_3DDEVICE | DDSCAPS_OFFSCREENPLAIN |
                      dwDeviceMemType;
hRes = lpdd->lpVtbl->CreateSurface(lpdd, &ddsd, &lpddDevice, NULL);
if (FAILED(hRes))
    return hRes;

// If we have created a palette, we have already determined that
// the primary surface (and hence the device surface) is palettized.
// Therefore, we should attach the palette to the device surface.
// (The palette is already attached to the primary surface.)

if (NULL != lpddPalette)
{
    hRes = lpddDevice->lpVtbl->SetPalette(lpddDevice, lpddPalette);
    if (FAILED(hRes))
        return hRes;
}

// We now determine whether or not we need a z-buffer and, if
```

```
// so, its bit depth.

if (0 != lpd3dDeviceDesc->dwDeviceZBufferBitDepth)
{
    // The device supports z-buffering. Determine the depth. We
    // select the lowest supported z-buffer depth to save memory.
    // (Accuracy is not too important for this sample.)

    dwZBufferBitDepth =
        FlagsToBitDepth(lpd3dDeviceDesc->dwDeviceZBufferBitDepth);

    // Create the z-buffer.

    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize            = sizeof(ddsd);
    ddsd.dwFlags           = DDSD_CAPS   |
                             DDSD_WIDTH  |
                             DDSD_HEIGHT |
                             DDSD_ZBUFFERBITDEPTH;
    ddsd.ddsCaps.dwCaps    = DDSCAPS_ZBUFFER | dwZBufferMemType;
    ddsd.dwWidth           = dwWidth;
    ddsd.dwHeight          = dwHeight;
    ddsd.dwZBufferBitDepth = dwZBufferBitDepth;
    hRes = lpdd->lpVtbl->CreateSurface(lpdd, &ddsd, &lpddZBuffer,
                                       NULL);
    if (FAILED(hRes))
        return hRes;

            // Attach it to the rendering target.

    hRes = lpddDevice->lpVtbl->AddAttachedSurface(lpddDevice,
                                                  lpddZBuffer);
    if (FAILED(hRes))
        return hRes;
}

// Now all the elements are in place: the device surface is in the
// correct memory type; a z-buffer has been attached with the
// correct depth and memory type; and a palette has been attached,
// if necessary. Now we can query for the Direct3D device we chose
// earlier.

hRes = lpddDevice->lpVtbl->QueryInterface(lpddDevice,
                                  &guidDevice, &lpd3dDevice);
if (FAILED(hRes))
    return hRes;
```

```
    return DD_OK;
}
```

Creating the Scene

The CreateScene function creates the elements making up the 3-D scene. In this sample, the scene consists of a single light, the viewport, the background and surface materials, the three transformation matrices, and the execute buffer holding the state changes and drawing primitives.

```
static HRESULT
CreateScene(void)
{
    HRESULT             hRes;
    D3DMATERIAL         d3dMaterial;
    D3DLIGHT            d3dLight;
    DWORD               dwVertexSize;
    DWORD               dwInstructionSize;
    DWORD               dwExecuteBufferSize;
    D3DEXECUTEBUFFERDESC d3dExecuteBufferDesc;
    D3DEXECUTEDATA      d3dExecuteData;

    ASSERT(NULL != lpd3d);
    ASSERT(NULL != lpd3dDevice);
    ASSERT(NULL == lpd3dViewport);
    ASSERT(NULL == lpd3dMaterial);
    ASSERT(NULL == lpd3dBackgroundMaterial);
    ASSERT(NULL == lpd3dExecuteBuffer);
    ASSERT(NULL == lpd3dLight);
    ASSERT(0  == hd3dWorldMatrix);
    ASSERT(0  == hd3dViewMatrix);
    ASSERT(0  == hd3dProjMatrix);

    // Create the light.

    hRes = lpd3d->lpVtbl->CreateLight(lpd3d, &lpd3dLight, NULL);
    if (FAILED(hRes))
        return hRes;

    ZeroMemory(&d3dLight, sizeof(d3dLight));
    d3dLight.dwSize = sizeof(d3dLight);
    d3dLight.dltType = D3DLIGHT_POINT;
    d3dLight.dcvColor.dvR    = D3DVAL( 1.0);
    d3dLight.dcvColor.dvG    = D3DVAL( 1.0);
```

```
d3dLight.dcvColor.dvB    = D3DVAL( 1.0);
d3dLight.dcvColor.dvA    = D3DVAL( 1.0);
d3dLight.dvPosition.dvX  = D3DVAL( 1.0);
d3dLight.dvPosition.dvY  = D3DVAL(-1.0);
d3dLight.dvPosition.dvZ  = D3DVAL(-1.0);
d3dLight.dvAttenuation0  = D3DVAL( 1.0);
d3dLight.dvAttenuation1  = D3DVAL( 0.1);
d3dLight.dvAttenuation2  = D3DVAL( 0.0);
hRes = lpd3dLight->lpVtbl->SetLight(lpd3dLight, &d3dLight);
if (FAILED(hRes))
   return hRes;

// Create the background material.

hRes = lpd3d->lpVtbl->CreateMaterial(lpd3d,
   &lpd3dBackgroundMaterial, NULL);
if (FAILED(hRes))
   return hRes;

ZeroMemory(&d3dMaterial, sizeof(d3dMaterial));
d3dMaterial.dwSize = sizeof(d3dMaterial);
d3dMaterial.dcvDiffuse.r  = D3DVAL(0.0);
d3dMaterial.dcvDiffuse.g  = D3DVAL(0.0);
d3dMaterial.dcvDiffuse.b  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.r  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.g  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.b  = D3DVAL(0.0);
d3dMaterial.dcvSpecular.r = D3DVAL(0.0);
d3dMaterial.dcvSpecular.g = D3DVAL(0.0);
d3dMaterial.dcvSpecular.b = D3DVAL(0.0);
d3dMaterial.dvPower       = D3DVAL(0.0);

// Since this is the background material, we don't want a ramp to be
// allocated. (We will not be smooth-shading the background.)

d3dMaterial.dwRampSize    = 1;

hRes = lpd3dBackgroundMaterial->lpVtbl->SetMaterial
   (lpd3dBackgroundMaterial, &d3dMaterial);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dBackgroundMaterial->lpVtbl->GetHandle
   (lpd3dBackgroundMaterial, lpd3dDevice, &hd3dBackgroundMaterial);
if (FAILED(hRes))
   return hRes;
```

```
// Create the viewport.
// The viewport parameters are set in the UpdateViewport function,
// which is called in response to WM_SIZE.

hRes = lpd3d->lpVtbl->CreateViewport(lpd3d, &lpd3dViewport, NULL);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dDevice->lpVtbl->AddViewport(lpd3dDevice, lpd3dViewport);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dViewport->lpVtbl->SetBackground(lpd3dViewport,
   hd3dBackgroundMaterial);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dViewport->lpVtbl->AddLight(lpd3dViewport, lpd3dLight);
if (FAILED(hRes))
   return hRes;

// Create the matrices.

hRes = lpd3dDevice->lpVtbl->CreateMatrix(lpd3dDevice,
   &hd3dWorldMatrix);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dDevice->lpVtbl->SetMatrix(lpd3dDevice, hd3dWorldMatrix,
   &d3dWorldMatrix);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dDevice->lpVtbl->CreateMatrix(lpd3dDevice,
   &hd3dViewMatrix);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dDevice->lpVtbl->SetMatrix(lpd3dDevice, hd3dViewMatrix,
   &d3dViewMatrix);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dDevice->lpVtbl->CreateMatrix(lpd3dDevice,
   &hd3dProjMatrix);
if (FAILED(hRes))
   return hRes;
SetPerspectiveProjection(&d3dProjMatrix, HALF_HEIGHT, FRONT_CLIP,
   BACK_CLIP);
hRes = lpd3dDevice->lpVtbl->SetMatrix(lpd3dDevice, hd3dProjMatrix,
   &d3dProjMatrix);
if (FAILED(hRes))
   return hRes;
```

```
// Create the surface material.

hRes = lpd3d->lpVtbl->CreateMaterial(lpd3d, &lpd3dMaterial, NULL);
if (FAILED(hRes))
   return hRes;
ZeroMemory(&d3dMaterial, sizeof(d3dMaterial));
d3dMaterial.dwSize = sizeof(d3dMaterial);

// Base green with white specular.

d3dMaterial.dcvDiffuse.r  = D3DVAL(0.0);
d3dMaterial.dcvDiffuse.g  = D3DVAL(1.0);
d3dMaterial.dcvDiffuse.b  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.r  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.g  = D3DVAL(0.4);
d3dMaterial.dcvAmbient.b  = D3DVAL(0.0);
d3dMaterial.dcvSpecular.r = D3DVAL(1.0);
d3dMaterial.dcvSpecular.g = D3DVAL(1.0);
d3dMaterial.dcvSpecular.b = D3DVAL(1.0);
d3dMaterial.dvPower       = D3DVAL(20.0);
d3dMaterial.dwRampSize    = 16;

hRes = lpd3dMaterial->lpVtbl->SetMaterial(lpd3dMaterial,
   &d3dMaterial);
if (FAILED(hRes))
   return hRes;

hRes = lpd3dMaterial->lpVtbl->GetHandle(lpd3dMaterial, lpd3dDevice,
   &hd3dSurfaceMaterial);
if (FAILED(hRes))
   return hRes;

// Build the execute buffer.

dwVertexSize      = (NUM_VERTICES       * sizeof(D3DVERTEX));
dwInstructionSize = (NUM_INSTRUCTIONS   * sizeof(D3DINSTRUCTION)) +
                    (NUM_STATES         * sizeof(D3DSTATE))      +
                    (NUM_PROCESSVERTICES *
                              sizeof(D3DPROCESSVERTICES))    +
                    (NUM_TRIANGLES      * sizeof(D3DTRIANGLE));
dwExecuteBufferSize = dwVertexSize + dwInstructionSize;
ZeroMemory(&d3dExecuteBufferDesc, sizeof(d3dExecuteBufferDesc));
d3dExecuteBufferDesc.dwSize       = sizeof(d3dExecuteBufferDesc);
d3dExecuteBufferDesc.dwFlags      = D3DDEB_BUFSIZE;
d3dExecuteBufferDesc.dwBufferSize = dwExecuteBufferSize;
```

```
    hRes = lpd3dDevice->lpVtbl->CreateExecuteBuffer(lpd3dDevice,
        &d3dExecuteBufferDesc, &lpd3dExecuteBuffer, NULL);
    if (FAILED(hRes))
        return hRes;

    // Fill the execute buffer with the required vertices, state
    // instructions and drawing primitives.

    hRes = FillExecuteBuffer();
    if (FAILED(hRes))
        return hRes;

    // Set the execute data so Direct3D knows how many vertices are in
    // the buffer and where the instructions start.

    ZeroMemory(&d3dExecuteData, sizeof(d3dExecuteData));
    d3dExecuteData.dwSize = sizeof(d3dExecuteData);
    d3dExecuteData.dwVertexCount       = NUM_VERTICES;
    d3dExecuteData.dwInstructionOffset = dwVertexSize;
    d3dExecuteData.dwInstructionLength = dwInstructionSize;
    hRes = lpd3dExecuteBuffer->lpVtbl->SetExecuteData
        (lpd3dExecuteBuffer, &d3dExecuteData);
    if (FAILED(hRes))
        return hRes;

    return DD_OK;
}
```

Filling the Execute Buffer

The **FillExecuteBuffer** function fills the single execute buffer used in this sample with all the vertices, transformations, light and render states, and drawing primitives necessary to draw our triangle.

The method shown here is not the most efficient way of organizing the execute buffer. For best performance you should minimize state changes. In this sample we submit the execute buffer for each frame in the animation loop and no state in the buffer is modified. The only thing we modify is the world matrix (its contents—not its handle). Therefore, it would be more efficient to extract all the static state instructions into a separate execute buffer which we would issue once only at startup and, from then on, simply execute a second execute buffer with vertices and triangles.

However, because this sample is more concerned with clarity than performance, it uses only one execute buffer_dx5_execute_buffer_glos and resubmits it in its entirety for each frame.

```
static HRESULT
```

```
FillExecuteBuffer(void)
{
    HRESULT             hRes;
    D3DEXECUTEBUFFERDESC d3dExeBufDesc;
    LPD3DVERTEX         lpVertex;
    LPD3DINSTRUCTION    lpInstruction;
    LPD3DPROCESSVERTICES lpProcessVertices;
    LPD3DTRIANGLE       lpTriangle;
    LPD3DSTATE          lpState;

    ASSERT(NULL != lpd3dExecuteBuffer);
    ASSERT(0  != hd3dSurfaceMaterial);
    ASSERT(0  != hd3dWorldMatrix);
    ASSERT(0  != hd3dViewMatrix);
    ASSERT(0  != hd3dProjMatrix);

    // Lock the execute buffer.

    ZeroMemory(&d3dExeBufDesc, sizeof(d3dExeBufDesc));
    d3dExeBufDesc.dwSize = sizeof(d3dExeBufDesc);
    hRes = lpd3dExecuteBuffer->lpVtbl->Lock(lpd3dExecuteBuffer,
        &d3dExeBufDesc);
    if (FAILED(hRes))
        return hRes;

    // For purposes of illustration, we fill the execute buffer by
    // casting a pointer to the execute buffer to the appropriate data
    // structures.

    lpVertex = (LPD3DVERTEX)d3dExeBufDesc.lpData;

    // First vertex.

    lpVertex->dvX  = D3DVAL( 0.0); // Position in model coordinates
    lpVertex->dvY  = D3DVAL( 1.0);
    lpVertex->dvZ  = D3DVAL( 0.0);
    lpVertex->dvNX = D3DVAL( 0.0); // Normalized illumination normal
    lpVertex->dvNY = D3DVAL( 0.0);
    lpVertex->dvNZ = D3DVAL(-1.0);
    lpVertex->dvTU = D3DVAL( 0.0); // Texture coordinates (not used)
    lpVertex->dvTV = D3DVAL( 1.0);
    lpVertex++;

    // Second vertex.

    lpVertex->dvX  = D3DVAL( 1.0); // Position in model coordinates
```

```
lpVertex->dvY  = D3DVAL(-1.0);
lpVertex->dvZ  = D3DVAL( 0.0);
lpVertex->dvNX = D3DVAL( 0.0); // Normalized illumination normal
lpVertex->dvNY = D3DVAL( 0.0);
lpVertex->dvNZ = D3DVAL(-1.0);
lpVertex->dvTU = D3DVAL( 1.0); // Texture coordinates (not used)
lpVertex->dvTV = D3DVAL( 1.0);
lpVertex++;

// Third vertex.

lpVertex->dvX  = D3DVAL(-1.0); // Position in model coordinates
lpVertex->dvY  = D3DVAL(-1.0);
lpVertex->dvZ  = D3DVAL( 0.0);
lpVertex->dvNX = D3DVAL( 0.0); // Normalized illumination normal
lpVertex->dvNY = D3DVAL( 0.0);
lpVertex->dvNZ = D3DVAL(-1.0);
lpVertex->dvTU = D3DVAL( 1.0); // Texture coordinates (not used)
lpVertex->dvTV = D3DVAL( 0.0);
lpVertex++;

// Transform state - world, view and projection.

lpInstruction = (LPD3DINSTRUCTION)lpVertex;
lpInstruction->bOpcode = D3DOP_STATETRANSFORM;
lpInstruction->bSize   = sizeof(D3DSTATE);
lpInstruction->wCount  = 3U;
lpInstruction++;
lpState = (LPD3DSTATE)lpInstruction;
lpState->dtstTransformStateType = D3DTRANSFORMSTATE_WORLD;
lpState->dwArg[0] = hd3dWorldMatrix;
lpState++;
lpState->dtstTransformStateType = D3DTRANSFORMSTATE_VIEW;
lpState->dwArg[0] = hd3dViewMatrix;
lpState++;
lpState->dtstTransformStateType = D3DTRANSFORMSTATE_PROJECTION;
lpState->dwArg[0] = hd3dProjMatrix;
lpState++;

// Lighting state.

lpInstruction = (LPD3DINSTRUCTION)lpState;
lpInstruction->bOpcode = D3DOP_STATELIGHT;
lpInstruction->bSize   = sizeof(D3DSTATE);
lpInstruction->wCount  = 2U;
lpInstruction++;
```

```
lpState = (LPD3DSTATE)lpInstruction;
lpState->dlstLightStateType = D3DLIGHTSTATE_MATERIAL;
lpState->dwArg[0] = hd3dSurfaceMaterial;
lpState++;
lpState->dlstLightStateType = D3DLIGHTSTATE_AMBIENT;
lpState->dwArg[0] = RGBA_MAKE(128, 128, 128, 128);
lpState++;

// Render state.

lpInstruction = (LPD3DINSTRUCTION)lpState;
lpInstruction->bOpcode = D3DOP_STATERENDER;
lpInstruction->bSize = sizeof(D3DSTATE);
lpInstruction->wCount = 3U;
lpInstruction++;
lpState = (LPD3DSTATE)lpInstruction;
lpState->drstRenderStateType = D3DRENDERSTATE_FILLMODE;
lpState->dwArg[0] = D3DFILL_SOLID;
lpState++;
lpState->drstRenderStateType = D3DRENDERSTATE_SHADEMODE;
lpState->dwArg[0] = D3DSHADE_GOURAUD;
lpState++;
lpState->drstRenderStateType = D3DRENDERSTATE_DITHERENABLE;
lpState->dwArg[0] = TRUE;
lpState++;

// The D3DOP_PROCESSVERTICES instruction tells the driver what to
// do with the vertices in the buffer. In this sample we want
// Direct3D to perform the entire pipeline on our behalf, so
// the instruction is D3DPROCESSVERTICES_TRANSFORMLIGHT.

lpInstruction = (LPD3DINSTRUCTION)lpState;
lpInstruction->bOpcode = D3DOP_PROCESSVERTICES;
lpInstruction->bSize   = sizeof(D3DPROCESSVERTICES);
lpInstruction->wCount  = 1U;
lpInstruction++;
lpProcessVertices = (LPD3DPROCESSVERTICES)lpInstruction;
lpProcessVertices->dwFlags    = D3DPROCESSVERTICES_TRANSFORMLIGHT;
lpProcessVertices->wStart     = 0U;          // First source vertex
lpProcessVertices->wDest      = 0U;
lpProcessVertices->dwCount    = NUM_VERTICES; // Number of vertices
lpProcessVertices->dwReserved = 0;
lpProcessVertices++;

// Draw the triangle.
```

```
    lpInstruction = (LPD3DINSTRUCTION)lpProcessVertices;
    lpInstruction->bOpcode = D3DOP_TRIANGLE;
    lpInstruction->bSize   = sizeof(D3DTRIANGLE);
    lpInstruction->wCount  = 1U;
    lpInstruction++;
    lpTriangle = (LPD3DTRIANGLE)lpInstruction;
    lpTriangle->wV1    = 0U;
    lpTriangle->wV2    = 1U;
    lpTriangle->wV3    = 2U;
    lpTriangle->wFlags = D3DTRIFLAG_EDGEENABLETRIANGLE;
    lpTriangle++;

    // Stop execution of the buffer.

    lpInstruction = (LPD3DINSTRUCTION)lpTriangle;
    lpInstruction->bOpcode = D3DOP_EXIT;
    lpInstruction->bSize   = 0;
    lpInstruction->wCount  = 0U;

    // Unlock the execute buffer.

    lpd3dExecuteBuffer->lpVtbl->Unlock(lpd3dExecuteBuffer);

    return DD_OK;
}
```

Animating the Scene

The animation in this sample is simply a rotation about the y-axis. All we need to do is build a rotation matrix and set the world matrix to that new rotation matrix.

We don't need to modify the execute buffer in any way to perform this rotation. We simply set the matrix and resubmit the execute buffer.

```
static HRESULT
AnimateScene(void)
{
    HRESULT hRes;

    ASSERT(NULL != lpd3dDevice);
    ASSERT(0  != hd3dWorldMatrix);

    // We rotate the triangle by setting the world transform to a
    // rotation matrix.

    SetRotationAboutY(&d3dWorldMatrix, dAngleOfRotation);
```

```
    dAngleOfRotation += ROTATE_ANGLE_DELTA;
    hRes = lpd3dDevice->lpVtbl->SetMatrix(lpd3dDevice,
        hd3dWorldMatrix, &d3dWorldMatrix);
    if (FAILED(hRes))
        return hRes;

    return DD_OK;
}
```

Rendering

This section contains functions that render the entire scene and render a single frame.

- Rendering the Scene
- Rendering a Single Frame

Rendering the Scene

The RenderScene function renders the 3-D scene, just as you might suspect. The fundamental task performed by this function is submitting the single execute buffer used by this sample. However, the function also clears the back and z-buffers and demarcates the start and end of the scene (which in this case is a single execute).

When you clear the back and z-buffers, it's safe to specify the z-buffer clear flag even if we don't have an attached z-buffer. Direct3D will simply discard the flag if no z-buffer is being used.

For maximum efficiency we only want to clear those regions of the device surface and z-buffer which we actually rendered to in the last frame. This is the purpose of the array of rectangles and count passed to this function. It is possible to query Direct3D for the regions of the device surface that were rendered to by that execute. The application can then accumulate those rectangles and clear only those regions. However this is a very simple sample and so, for simplicity, we will just clear the entire device surface and z-buffer. You should probably implement a more efficient clearing mechanism in your application.

The RenderScene function must be called once and once only for every frame of animation. If you have multiple execute buffers comprising a single frame you must have one call to the **IDirect3DDevice2::BeginScene** method before submitting those execute buffers. If you have more than one device being rendered in a single frame, (for example, a rear-view mirror in a racing game), call the **IDirect3DDevice::BeginScene** and **IDirect3DDevice2::EndScene** methods once for each device.

When the RenderScene function returns DD_OK, the scene will have been rendered and the device surface will hold the contents of the rendering.

```
static HRESULT
```

```
RenderScene(void)
{
    HRESULT hRes;
    D3DRECT d3dRect;

    ASSERT(NULL != lpd3dViewport);
    ASSERT(NULL != lpd3dDevice);
    ASSERT(NULL != lpd3dExecuteBuffer);

    // Clear both back and z-buffer.

    d3dRect.lX1 = rSrcRect.left;
    d3dRect.lX2 = rSrcRect.right;
    d3dRect.lY1 = rSrcRect.top;
    d3dRect.lY2 = rSrcRect.bottom;
    hRes = lpd3dViewport->lpVtbl->Clear(lpd3dViewport, 1, &d3dRect,
        D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER);
    if (FAILED(hRes))
        return hRes;

    // Start the scene.

    hRes = lpd3dDevice->lpVtbl->BeginScene(lpd3dDevice);
    if (FAILED(hRes))
        return hRes;

    // Submit the execute buffer.

    // We want Direct3D to clip the data on our behalf so we specify
    // D3DEXECUTE_CLIPPED.

    hRes = lpd3dDevice->lpVtbl->Execute(lpd3dDevice, lpd3dExecuteBuffer,
        lpd3dViewport, D3DEXECUTE_CLIPPED);
    if (FAILED(hRes))
    {
        lpd3dDevice->lpVtbl->EndScene(lpd3dDevice);
        return hRes;
    }

        // End the scene.

    hRes = lpd3dDevice->lpVtbl->EndScene(lpd3dDevice);
    if (FAILED(hRes))
        return hRes;

    return DD_OK;
```

```
}
```

Rendering a Single Frame

The DoFrame function renders and shows a single frame. This involves rendering the scene and blitting the result to the client area of the application window on the primary surface.

This function handles lost surfaces by attempting to restore the application's surfaces and then retrying the rendering. It is called by the OnMove function (discussed in Redrawing on Window Movement), the OnSize function (discussed in Redrawing on Window Resizing), and the OnPaint function (discussed in Repainting the Client Area).

```
static HRESULT
DoFrame(void)
{
    HRESULT hRes;

    // We keeping trying until we succeed or we fail for a reason
    // other than DDERR_SURFACELOST.

    while (TRUE)
    {
        hRes = RenderScene();
        if (SUCCEEDED(hRes))
        {
            hRes = lpddPrimary->lpVtbl->Blt(lpddPrimary, &rDstRect,
                lpddDevice, &rSrcRect, DDBLT_WAIT, NULL);
            if (SUCCEEDED(hRes))  // If it worked.
                return hRes;
        }
        while (DDERR_SURFACELOST == hRes)  // Restore lost surfaces
            hRes = RestoreSurfaces();
        if (FAILED(hRes))  // handle other failure cases
            return hRes;
    }
}
```

Working with Matrices

This section contains two functions that work with matrices: the SetPerspectiveProjection function, which sets a given matrix to the appropriate values for the front and back clipping plane, and the SetRotationAboutY function, which sets a matrix to a rotation about the y-axis.

- Setting the Perspective Transformation
- Setting a Rotation Transformation

Setting the Perspective Transformation

The SetPerspectiveProjection function sets the given matrix to a perspective transform for the given half-height and front- and back-clipping planes. This function is called as part of the CreateScene function, documented in Creating the Scene.

```
static void
SetPerspectiveProjection(LPD3DMATRIX lpd3dMatrix,
                         double      dHalfHeight,
                         double      dFrontClipping,
                         double      dBackClipping)
{
    double dTmp1;
    double dTmp2;

    ASSERT(NULL != lpd3dMatrix);

    dTmp1 = dHalfHeight / dFrontClipping;
    dTmp2 = dBackClipping / (dBackClipping - dFrontClipping);

    lpd3dMatrix->_11 =  D3DVAL(2.0);
    lpd3dMatrix->_12 =  D3DVAL(0.0);
    lpd3dMatrix->_13 =  D3DVAL(0.0);
    lpd3dMatrix->_14 =  D3DVAL(0.0);
    lpd3dMatrix->_21 =  D3DVAL(0.0);
    lpd3dMatrix->_22 =  D3DVAL(2.0);
    lpd3dMatrix->_23 =  D3DVAL(0.0);
    lpd3dMatrix->_24 =  D3DVAL(0.0);
    lpd3dMatrix->_31 =  D3DVAL(0.0);
    lpd3dMatrix->_32 =  D3DVAL(0.0);
    lpd3dMatrix->_33 =  D3DVAL(dTmp1 * dTmp2);
    lpd3dMatrix->_34 =  D3DVAL(dTmp1);
    lpd3dMatrix->_41 =  D3DVAL(0.0);
    lpd3dMatrix->_42 =  D3DVAL(0.0);
    lpd3dMatrix->_43 =  D3DVAL(-dHalfHeight * dTmp2);
    lpd3dMatrix->_44 =  D3DVAL(0.0);
}
```

Setting a Rotation Transformation

The SetRotationAboutY function sets the given matrix to a rotation about the y-axis, using the specified number of radians. This function is called as part of the AnimateScene function, documented in Animating the Scene.

```
static void
SetRotationAboutY(LPD3DMATRIX lpd3dMatrix, double dAngleOfRotation)
{
    D3DVALUE dvCos;
    D3DVALUE dvSin;

    ASSERT(NULL != lpd3dMatrix);

    dvCos = D3DVAL(cos(dAngleOfRotation));
    dvSin = D3DVAL(sin(dAngleOfRotation));

    lpd3dMatrix->_11 =  dvCos;
    lpd3dMatrix->_12 =  D3DVAL(0.0);
    lpd3dMatrix->_13 = -dvSin;
    lpd3dMatrix->_14 =  D3DVAL(0.0);
    lpd3dMatrix->_21 =  D3DVAL(0.0);
    lpd3dMatrix->_22 =  D3DVAL(1.0);
    lpd3dMatrix->_23 =  D3DVAL(0.0);
    lpd3dMatrix->_24 =  D3DVAL(0.0);
    lpd3dMatrix->_31 =  dvSin;
    lpd3dMatrix->_32 =  D3DVAL(0.0);
    lpd3dMatrix->_33 =  dvCos;
    lpd3dMatrix->_34 =  D3DVAL(0.0);
    lpd3dMatrix->_41 =  D3DVAL(0.0);
    lpd3dMatrix->_42 =  D3DVAL(0.0);
    lpd3dMatrix->_43 =  D3DVAL(0.0);
    lpd3dMatrix->_44 =  D3DVAL(1.0);
}
```

Restoring and Redrawing

This section contains functions that restore objects and surfaces that may have been lost while the application is running.

- Restoring the Direct3D Device
- Restoring the Primary Surface
- Restoring All Surfaces
- Redrawing on Window Movement
- Redrawing on Window Resizing
- Repainting the Client Area

- Updating the Viewport

Restoring the Direct3D Device

The RestoreDevice function restores lost video memory for the device surface and z-buffer.

```
static HRESULT
RestoreDevice(void)
{
    HRESULT hRes;

    if (NULL != lpddZBuffer)
    {
        hRes = lpddZBuffer->lpVtbl->Restore(lpddZBuffer);
        if (FAILED(hRes))
            return hRes;
    }

    if (NULL != lpddDevice)
    {
        hRes = lpddDevice->lpVtbl->Restore(lpddDevice);
        if (FAILED(hRes))
            return hRes;
    }

    return DD_OK;
}
```

Restoring the Primary Surface

The RestorePrimary function attempts to restore the video memory allocated for the primary surface. This function will be invoked by a DirectX function returning DDERR_SURFACELOST due to a mode switch or full-screen DOS box invalidating video memory.

```
static HRESULT
RestorePrimary(void)
{
    ASSERT(NULL != lpddPrimary);

    return lpddPrimary->lpVtbl->Restore(lpddPrimary);
}
```

Restoring All Surfaces

The RestoreSurfaces function attempts to restore all the surfaces used by the application.

```
static LRESULT
RestoreSurfaces(void)
{
    HRESULT hRes;

    hRes = RestorePrimary();
    if (FAILED(hRes))
        return hRes;

    hRes = RestoreDevice();
    if (FAILED(hRes))
        return hRes;

    return DD_OK;
}
```

Redrawing on Window Movement

```
static LRESULT
OnMove(HWND hwnd, int x, int y)
{
    int     xDelta;
    int     yDelta;
    HRESULT hRes;

    // No action if the device has not yet been created or if we are
    // suspended.

    if ((NULL != lpd3dDevice) && !fSuspended)
    {
        // Update the destination rectangle for the new client position.

        xDelta = x - rDstRect.left;
        yDelta = y - rDstRect.top;

        rDstRect.left   += xDelta;
        rDstRect.top    += yDelta;
        rDstRect.right  += xDelta;
        rDstRect.bottom += yDelta;

        // Repaint the client area.
```

```
        hRes = DoFrame();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_RENDERSCENE, hRes);
            return 0L;
        }
    }

    return 0L;
}
```

Redrawing on Window Resizing

```
static LRESULT
OnSize(HWND hwnd, int w, int h)
{
    HRESULT       hRes;
    DDSURFACEDESC ddsd;

        // Nothing to do if we are suspended.

    if (!fSuspended)
    {
        // Update the source and destination rectangles (used by the
        // blit that shows the rendering in the client area).

        rDstRect.right  = rDstRect.left + w;
        rDstRect.bottom = rDstRect.top  + h;
        rSrcRect.right  = w;
        rSrcRect.bottom = h;

        if (NULL != lpd3dDevice)
        {
            // Although we already have a device, we need to be sure it
            // is big enough for the new window client size.

            // Because the window in this sample has a fixed size, it
            // should never be necessary to handle this case. This code
            // will be useful when we make the application resizable.

            ZeroMemory(&ddsd, sizeof(ddsd));
            ddsd.dwSize = sizeof(ddsd);
            hRes = lpddDevice->lpVtbl->GetSurfaceDesc(lpddDevice,
                &ddsd);
            if (FAILED(hRes))
```

```
        {
            FatalError(hwnd, IDS_ERRMSG_DEVICESIZE, hRes);
            return 0L;
        }

        if ((w > (int)ddsd.dwWidth) || (h > (int)ddsd.dwHeight))
        {
            // The device is too small. We need to shut it down
            // and rebuild it.

            // Execute buffers are bound to devices, so when
            // we release the device we must release the execute
            // buffer.

            ReleaseScene();
            ReleaseDevice();
        }
    }

    if (NULL == lpd3dDevice)
    {
        // No Direct3D device yet. This is either because this is
        // the first time through the loop or because we discarded
        // the existing device because it was not big enough for the
        // new window client size.

        hRes = CreateDevice((DWORD)w, (DWORD)h);
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_CREATEDEVICE, hRes);
            return 0L;
        }
        hRes = CreateScene();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_BUILDSCENE, hRes);
            return 0L;
        }
    }

    hRes = UpdateViewport();
    if (FAILED(hRes))
    {
        FatalError(hwnd, IDS_ERRMSG_UPDATEVIEWPORT, hRes);
        return 0L;
    }
```

```
        // Render at the new size and show the results in the window's
        // client area.

        hRes = DoFrame();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_RENDERSCENE, hRes);
            return 0L;
        }
    }

    return 0L;
}
```

Repainting the Client Area

The OnPaint function repaints the client area, when required. Notice that it calls the DoFrame function to do much of the work, even though DoFrame re-renders the scene as well as blitting the result to the primary surface. Although the re-rendering is not necessary, for this simple sample this inefficiency does not matter. In your application, you should re-render only when the scene changes.

For more information about the DoFrame function, see Rendering a Single Frame.

```
static LRESULT
OnPaint(HWND hwnd, HDC hdc, LPPAINTSTRUCT lpps)
{
    HRESULT hRes;

    USE_PARAM(lpps);

    if (fActive && !fSuspended && (NULL != lpd3dDevice))
    {
        hRes = DoFrame();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_RENDERSCENE, hRes);
            return 0L;
        }
    }
    else
    {
        // Show the suspended image if we are not active or suspended or
        // if we have not yet created the device.
```

```
        PaintSuspended(hwnd, hdc);
    }

    return 0L;
}
```

Updating the Viewport

The UpdateViewport function updates the viewport in response to a change in window size. This ensures that we render at a resolution that matches the client area of the target window.

```
static HRESULT
UpdateViewport(void)
{
    D3DVIEWPORT d3dViewport;

    ASSERT(NULL != lpd3dViewport);

    ZeroMemory(&d3dViewport, sizeof(d3dViewport));
    d3dViewport.dwSize   = sizeof(d3dViewport);
    d3dViewport.dwX      = 0;
    d3dViewport.dwY      = 0;
    d3dViewport.dwWidth  = (DWORD)rSrcRect.right;
    d3dViewport.dwHeight = (DWORD)rSrcRect.bottom;
    d3dViewport.dvScaleX = D3DVAL((float)d3dViewport.dwWidth / 2.0);
    d3dViewport.dvScaleY = D3DVAL((float)d3dViewport.dwHeight / 2.0);
    d3dViewport.dvMaxX   = D3DVAL(1.0);
    d3dViewport.dvMaxY   = D3DVAL(1.0);
    return lpd3dViewport->lpVtbl->SetViewport(lpd3dViewport,
        &d3dViewport);
}
```

Releasing Objects

This section contains functions that release objects when they are no longer needed.

- Releasing the Direct3D Object
- Releasing the Direct3D Device
- Releasing the Primary Surface
- Releasing the Objects in the Scene

Releasing the Direct3D Object

The ReleaseDirect3D function releases the DirectDraw (Direct3D) driver object.

```
static HRESULT
ReleaseDirect3D(void)
{
    if (NULL != lpd3d)
    {
        lpd3d->lpVtbl->Release(lpd3d);
        lpd3d = NULL;
    }
    if (NULL != lpdd)
    {
        lpdd->lpVtbl->Release(lpdd);
        lpdd = NULL;
    }

    return DD_OK;
}
```

Releasing the Direct3D Device

The ReleaseDevice function releases the Direct3D device and its associated surfaces.

```
static HRESULT
ReleaseDevice(void)
{
    if (NULL != lpd3dDevice)
    {
        lpd3dDevice->lpVtbl->Release(lpd3dDevice);
        lpd3dDevice = NULL;
    }
    if (NULL != lpddZBuffer)
    {
        lpddZBuffer->lpVtbl->Release(lpddZBuffer);
        lpddZBuffer = NULL;
    }
    if (NULL != lpddDevice)
    {
        lpddDevice->lpVtbl->Release(lpddDevice);
        lpddDevice = NULL;
    }

    return DD_OK;
}
```

Releasing the Primary Surface

The ReleasePrimary function releases the primary surface and its attached clipper and palette.

```
static HRESULT
ReleasePrimary(void)
{
    if (NULL != lpddPalette)
    {
        lpddPalette->lpVtbl->Release(lpddPalette);
        lpddPalette = NULL;
    }
    if (NULL != lpddPrimary)
    {
        lpddPrimary->lpVtbl->Release(lpddPrimary);
        lpddPrimary = NULL;
    }

    return DD_OK;
}
```

Releasing the Objects in the Scene

The ReleaseScene function releases all the objects making up the 3-D scene.

```
static HRESULT
ReleaseScene(void)
{
    if (NULL != lpd3dExecuteBuffer)
    {
        lpd3dExecuteBuffer->lpVtbl->Release(lpd3dExecuteBuffer);
        lpd3dExecuteBuffer = NULL;
    }
    if (NULL != lpd3dBackgroundMaterial)
    {
        lpd3dBackgroundMaterial->
            lpVtbl->Release(lpd3dBackgroundMaterial);
        lpd3dBackgroundMaterial = NULL;
    }
    if (NULL != lpd3dMaterial)
    {
        lpd3dMaterial->lpVtbl->Release(lpd3dMaterial);
        lpd3dMaterial = NULL;
    }
    if (0 != hd3dWorldMatrix)
```

```
    {
        lpd3dDevice->lpVtbl->DeleteMatrix(lpd3dDevice, hd3dWorldMatrix);
        hd3dWorldMatrix = 0;
    }
    if (0 != hd3dViewMatrix)
    {
        lpd3dDevice->lpVtbl->DeleteMatrix(lpd3dDevice, hd3dViewMatrix);
        hd3dViewMatrix = 0;
    }
    if (0 != hd3dProjMatrix)
    {
        lpd3dDevice->lpVtbl->DeleteMatrix(lpd3dDevice, hd3dProjMatrix);
        hd3dProjMatrix = 0;
    }
    if (NULL != lpd3dLight)
    {
        lpd3dLight->lpVtbl->Release(lpd3dLight);
        lpd3dLight = NULL;
    }
    if (NULL != lpd3dViewport)
    {
        lpd3dViewport->lpVtbl->Release(lpd3dViewport);
        lpd3dViewport = NULL;
    }

    return DD_OK;
}
```

Error Checking

This section contains functions that help you check for and report errors.

- Checking for Active Status
- Reporting Standard Errors
- Reporting Fatal Errors
- Displaying a Notification String

Checking for Active Status

```
static LRESULT
OnIdle(HWND hwnd)
{
    HRESULT hRes;
```

```
    // Only animate if we are the foreground app, we aren't suspended,
    // and we have completed initialization.

    if (fActive && !fSuspended && (NULL != lpd3dDevice))
    {
        hRes = AnimateScene();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_ANIMATESCENE, hRes);
            return 0L;
        }

        hRes = DoFrame();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_RENDERSCENE, hRes);
            return 0L;
        }
    }

    return 0L;
}
```

Reporting Standard Errors

The ReportError function displays a message box to report an error.

```
static void
ReportError(HWND hwnd, int nMessage, HRESULT hRes)
{
    HDC  hdc;
    char szBuffer[256];
    char szMessage[128];
    char szError[128];
    int  nStrID;

    // Turn the animation loop off.

    fSuspended = TRUE;

    // Get the high level error message.

    LoadString(hAppInstance, nMessage, szMessage, sizeof(szMessage));

    // We issue sensible error messages for common run time errors. For
```

```
    // errors which are internal or coding errors we simply issue an
    // error number (they should never occur).

    switch (hRes)
    {
        case DDERR_EXCEPTION:        nStrID = IDS_ERR_EXCEPTION;        break;
        case DDERR_GENERIC:          nStrID = IDS_ERR_GENERIC;          break;
        case DDERR_OUTOFMEMORY:      nStrID = IDS_ERR_OUTOFMEMORY;      break;
        case DDERR_OUTOFVIDEOMEMORY: nStrID = IDS_ERR_OUTOFVIDEOMEMORY;
break;
        case DDERR_SURFACEBUSY:      nStrID = IDS_ERR_SURFACEBUSY;      break;
        case DDERR_SURFACELOST:      nStrID = IDS_ERR_SURFACELOST;      break;
        case DDERR_WRONGMODE:        nStrID = IDS_ERR_WRONGMODE;        break;
        default:                     nStrID = IDS_ERR_INTERNALERROR;    break;
    }
    LoadString(hAppInstance, nStrID, szError, sizeof(szError));

    // Show the "paused" display.

    hdc = GetDC(hwnd);
    PaintSuspended(hwnd, hdc);
    ReleaseDC(hwnd, hdc);

    // Convert the error code into a string.

    wsprintf(szBuffer, "%s\n%s (Error #%d)", szMessage, szError,
        CODEFROMHRESULT(hRes));
    MessageBox(hwnd, szBuffer, WINDOW_TITLE, MB_OK | MB_APPLMODAL);
    fSuspended = FALSE;
}
```

Reporting Fatal Errors

The FatalError function displays a message box to report an error message and then destroys the window. The function does not perform any clean-up; this is done when the application receives the WM_DESTROY message sent by the **DestroyWindow** function.

```
static void
FatalError(HWND hwnd, int nMessage, HRESULT hRes)
{
    ReportError(hwnd, nMessage, hRes);
    fSuspended = TRUE;

    DestroyWindow(hwnd);
}
```

Displaying a Notification String

The PaintSuspended function draws a notification string in the client area whenever the application is suspended—for example, when it is in the background or is handling an error.

```
static void
PaintSuspended(HWND hwnd, HDC hdc)
{
   HPEN     hOldPen;
   HBRUSH   hOldBrush;
   COLORREF crOldTextColor;
   int      oldMode;
   int      x;
   int      y;
   SIZE     size;
   RECT     rect;
   int      nStrLen;

   // Black background.

   hOldPen   = SelectObject(hdc, GetStockObject(NULL_PEN));
   hOldBrush = SelectObject(hdc, GetStockObject(BLACK_BRUSH));

   // White text.

   oldMode = SetBkMode(hdc, TRANSPARENT);
   crOldTextColor = SetTextColor(hdc, RGB(255, 255, 255));

   GetClientRect(hwnd, &rect);

   // Clear the client area.

   Rectangle(hdc, rect.left, rect.top, rect.right + 1, rect.bottom + 1);

   // Draw the string centered in the client area.

   nStrLen = strlen(PAUSED_STRING);
   GetTextExtentPoint32(hdc, PAUSED_STRING, nStrLen, &size);
   x = (rect.right  - size.cx) / 2;
   y = (rect.bottom - size.cy) / 2;
   TextOut(hdc, x, y, PAUSED_STRING, nStrLen);

   SetTextColor(hdc, crOldTextColor);
   SetBkMode(hdc, oldMode);
```

```
    SelectObject(hdc, hOldBrush);
    SelectObject(hdc, hOldPen);
}
```

Converting Bit Depths

This section contains functions that convert bit depths into flags and vice versa.

- Converting a Bit Depth into a Flag
- Converting a Flag into a Bit Depth

Converting a Bit Depth into a Flag

The BitDepthToFlags function is used by the ChooseDevice enumeration function to convert a bit depth into the appropriate DirectDraw bit depth flag. For more information, see Enumeration Function

```
static DWORD
BitDepthToFlags(DWORD dwBitDepth)
{
    switch (dwBitDepth)
    {
        case  1: return DDBD_1;
        case  2: return DDBD_2;
        case  4: return DDBD_4;
        case  8: return DDBD_8;
        case 16: return DDBD_16;
        case 24: return DDBD_24;
        case 32: return DDBD_32;
        default: return 0;
    }
}
```

Converting a Flag into a Bit Depth

The FlagsToBitDepth function is used by the CreateDevice function to convert bit-depth flags to an actual bit count. It selects the smallest bit count in the mask if more than one flag is present. For more information, see Creating the Direct3D Device.

```
static DWORD
FlagsToBitDepth(DWORD dwFlags)
{
    if (dwFlags & DDBD_1)
        return 1;
```

```
    else if (dwFlags & DDBD_2)
        return 2;
    else if (dwFlags & DDBD_4)
        return 4;
    else if (dwFlags & DDBD_8)
        return 8;
    else if (dwFlags & DDBD_16)
        return 16;
    else if (dwFlags & DDBD_24)
        return 24;
    else if (dwFlags & DDBD_32)
        return 32;
    else
        return 0;
}
```

Main Window Procedure

```
LRESULT CALLBACK
WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    HDC         hdc;
    PAINTSTRUCT ps;
    LRESULT     lResult;
    HRESULT     hRes;
    char        szBuffer[128];

    switch (msg)
    {
        case WM_CREATE:
            hRes = CreateDirect3D(hwnd);
            if (FAILED(hRes))
            {
                ReportError(hwnd, IDS_ERRMSG_CREATEDEVICE, hRes);
                ReleaseDirect3D();
                return -1L;
            }

            hRes = CreatePrimary(hwnd);
            if (FAILED(hRes))
            {
                ReportError(hwnd, IDS_ERRMSG_INITSCREEN, hRes);
                ReleasePrimary();
                ReleaseDirect3D();
                return -1L;
```

```
    }

    hRes = ChooseDevice();
    if (FAILED(hRes))
    {
        ReportError(hwnd, IDS_ERRMSG_NODEVICE, hRes);
        ReleasePrimary();
        ReleaseDirect3D();
        return -1L;
    }

    // Update the title to show the name of the chosen device.

    wsprintf(szBuffer, "%s: %s", WINDOW_TITLE, szDeviceName);
    SetWindowText(hwnd, szBuffer);

    return 0L;

case WM_MOVE:
    return OnMove(hwnd, (int)LOWORD(lParam),
        (int)HIWORD(lParam));

case WM_SIZE:
    return OnSize(hwnd, (int)LOWORD(lParam),
        (int)HIWORD(lParam));

case WM_ERASEBKGND:
// Our rendering fills the entire viewport so we won't bother
// erasing the background.

    return 1L;

case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);

    lResult = OnPaint(hwnd, hdc, &ps);

    EndPaint(hwnd, &ps);
    return lResult;

case WM_ACTIVATEAPP:
    fActive = (BOOL)wParam;
    if (fActive && !fSuspended && (NULL != lpddPalette))
    {
        // Realizing the palette using DirectDraw is different
        // from GDI. To realize the palette we call SetPalette
```

```
            // each time our application is activated.

            // NOTE: DirectDraw recognizes that the new palette
            // is the same as the old one and so does not increase
            // the reference count of the palette.

            hRes = lpddPrimary->lpVtbl->SetPalette(lpddPrimary,
               lpddPalette);
            if (FAILED(hRes))
            {
               FatalError(hwnd, IDS_ERRMSG_REALIZEPALETTE, hRes);
               return 0L;
            }

         }
         else
         {
            // If we have been deactived, invalidate to show
            // the suspended display.

            InvalidateRect(hwnd, NULL, FALSE);
         }
         return 0L;

      case WM_KEYUP:
         // We use the escape key as a quick way of
         // getting out of the application.

         if (VK_ESCAPE == (int)wParam)
         {
            DestroyWindow(hwnd);
            return 0L;
         }
         break;

      case WM_CLOSE:
         DestroyWindow(hwnd);
         return 0L;

      case WM_DESTROY:
         // All cleanup is done here when terminating normally or
         // shutting down due to an error.

         ReleaseScene();
         ReleaseDevice();
         ReleasePrimary();
```

```
        ReleaseDirect3D();

        PostQuitMessage(0);
        return 0L;
    }

    return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

WinMain Function

```
int PASCAL
WinMain(HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPSTR     lpszCommandLine,
        int       cmdShow)
{
    WNDCLASS wndClass;
    HWND     hwnd;
    MSG      msg;

    USE_PARAM(hPrevInstance);

    // Record the instance handle.

    hAppInstance = hInstance;

    // Very simple command-line processing. We only have one
    // option - debug - so we will just assume that if anything was
    // specified on the command line the user wants debug mode.
    // (In debug mode there is no hardware and all surfaces are
    // explicitly in system memory.)

    if (0 != *lpszCommandLine)
        fDebug = TRUE;

    // Register the window class.

    wndClass.style         = 0;
    wndClass.lpfnWndProc   = WndProc;
    wndClass.cbClsExtra    = 0;
    wndClass.cbWndExtra    = 0;
    wndClass.hInstance     = hInstance;
    wndClass.hIcon         = LoadIcon(hAppInstance,
        MAKEINTRESOURCE(IDI_APPICON));
```

```
wndClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
wndClass.hbrBackground = GetStockObject(WHITE_BRUSH);
wndClass.lpszMenuName  = NULL;
wndClass.lpszClassName = WINDOW_CLASSNAME;

RegisterClass(&wndClass);

// Create the main window of the instance.

hwnd = CreateWindow(WINDOW_CLASSNAME,
                    WINDOW_TITLE,
                    WS_OVERLAPPED | WS_SYSMENU,
                    CW_USEDEFAULT, CW_USEDEFAULT,
                    WINDOW_WIDTH, WINDOW_HEIGHT,
                    NULL,
                    NULL,
                    hInstance,
                    NULL);

ShowWindow(hwnd, cmdShow);
UpdateWindow(hwnd);

// The main message dispatch loop.

// NOTE: For simplicity we handle the message loop with a
// simple PeekMessage scheme. This might not be the best
// mechanism for a real application (a separate render worker
// thread might be better).

while (TRUE)
{
   if (PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE))
   {
      // Message pending. If it's QUIT then exit the message
      // loop. Otherwise, process the message.

      if (WM_QUIT == msg.message)
      {
         break;
      }
      else
      {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
      }
   }
```

```
        else
        {
            // Animate the scene.

            OnIdle(hwnd);
        }
    }

    return msg.wParam;
}
```