Part A

Microsoft® DirectX™ 2 Software Development Kit

C H A P T E R 5

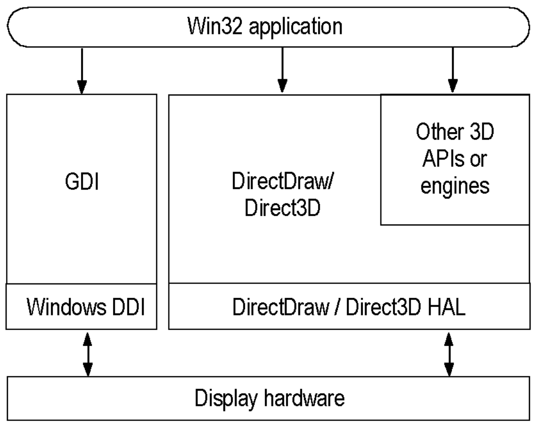
Direct3D Overviews

Direct3D Overview

Microsoft's 3D-Graphics Solutions

The Microsoft family of advanced 3D-graphics solutions includes the Direct3D™ and OpenGL® application programming interfaces (APIs).

The relationship between the Windows graphics systems, applications written with the Win32® API, the rest of the Windows system, and the hardware is shown in the following illustration.



Direct3D

Direct3D is Microsoft's high-speed 3D software solution. It gives you the ability to create high-performance, real-time 3D applications for typical desktop computers. The system was designed for speed and requires little memory.

Direct3D is implemented in two distinctly different modes: Retained Mode, a high-level API in which the application retains the graphics data, and Immediate Mode, a low-level API in which the application explicitly streams the data out to an execute buffer.

Retained Mode

Direct3D Retained Mode API is designed for manipulating 3D objects and managing 3D scenes. Retained Mode makes it easy to add 3D capabilities to existing Windows-based applications or to create new 3D applications. Its built-in geometry engine supports advanced capabilities like key frame animation and frees you from creating object databases and managing the internal structures of objects. In other words, after using a single call to load a predefined 3D object, the application can use simple methods from the API to manipulate the object in the scene in real-time without having to work explicitly with any of the internal geometry.

Retained Mode is built on top of Immediate Mode and is tightly coupled with the DirectDraw™ application programming interface (API). Microsoft will incorporate Retained Mode into a future version of Windows. For more information, see **DirectDraw** and **Introduction to Direct3D Retained-Mode Objects**.

Immediate Mode

Direct3D Immediate Mode is Microsoft's low-level 3D API. It allows you to port games and other high-performance multimedia applications to the Windows operating system.

Immediate Mode is a thin layer above real-time 3D accelerator hardware that gives you access to the features of that hardware. It also offers optimal software rendering for some hardware features that are not present. Immediate Mode gives you the flexibility to exploit your own rendering and scene management technologies. It is a device-independent way for applications to communicate with accelerator hardware at a low level, enabling maximum performance.

Unlike Retained Mode, Immediate Mode does not provide a geometry engine; applications that use Immediate Mode must provide their own object and scene management. Therefore, you should be knowledgeable in 3D graphics programming to use Immediate Mode effectively.

Direct3D is based on the OLE Component Object Model (COM), and is tightly integrated with **DirectDraw**. Microsoft will incorporate Direct3D into a future version of Windows. For more information, see **Introduction to Direct3D Immediate-Mode Objects**.

Hardware Abstraction and Emulation

The Direct3D API (like the rest of the DirectX API) is built on top of a hardware-abstraction layer (HAL), which insulates you from device-specific dependencies present in the hardware. A companion piece to the Direct3D HAL is the hardware-emulation layer (HEL). The Direct3D HEL provides software-based emulation of features that are not present in hardware. The combination of these hardware abstraction and emulation layers ensures that the services of the API are always available to you.

The Direct3D HAL is tightly integrated with the DirectDraw HAL and the GDI display driver, giving hardware manufacturers a single, consistent interface to Microsoft graphics APIs, and a long-term, unified driver model for accelerated 3D. Hardware manufacturers need to write only a single driver to accelerate Direct3D, DirectDraw, GDI, and OpenGL. Hardware can accelerate all or part of the 3D graphics rendering pipeline, including geometry transformations, 3D clipping and lighting, and rasterization. The Direct3D HAL has been designed to accommodate future graphics accelerators in addition to those available today.

DirectDraw

DirectDraw provides the fastest way to display graphics on-screen. It is the Windows composition engine for 2D graphics, 3D graphics, and video. DirectDraw composes and moves images very quickly and can employ page flipping to enable smooth animation. This combination of capabilities allows you to create high-speed games and multimedia applications and to port existing titles to Windows quickly and easily. DirectDraw is also the composition engine for all of Microsoft's newest graphics subsystems. You can use it to quickly integrate images generated by Windows GDI, Direct3D, ActiveMovie™, and OpenGL.

DirectDraw is a thin layer above the display hardware that enables you to easily take advantage of the powerful composition abilities of graphics accelerators designed for Windows, including high-speed blitting, interpolated stretching, and overlays. It also supports color space conversion, allowing accelerated video playback. Like Direct3D, DirectDraw is a device-independent way for applications to communicate with hardware. Under MS-DOS, you had to optimize code for each target device. With DirectDraw, however, you can attain high performance consistently across all of the hardware that accelerates DirectDraw.

DirectDraw is a COM-based API. Microsoft will incorporate DirectDraw into a future version of Windows. For more information, see **About DirectDraw**.

OpenGL

OpenGL is a precise 3D technology used for high-end CAD/CAM, modeling and animation, simulations, scientific visualization, and other exacting 3D-image rendering. It is provided with Windows NT and is available to you for use with Windows 95. Putting OpenGL on Windows 95 means that you can run Win32 OpenGL applications on any Win32 workstation. OpenGL currently takes advantage of any high-end hardware that has OpenGL functionality, using a client driver model designed for OpenGL. A future release of OpenGL will be able to take advantage of lower-priced 3D hardware (provided that it supports the precision conformance requirements of OpenGL) through the Direct3D API, providing a hardware solution that complements Direct3D.

Direct3D Architecture

The Direct3D Vision

Direct3D is designed to enable world-class game and interactive 3D graphics on a computer running Windows. Its mission is to provide device-dependent access to 3D video-display hardware in a device-independent manner. The application does not need to implement the specific calling procedures required to draw texture-mapped, perspective-corrected, or alpha-blended 3D primitives on a particular piece of hardware. Simply put, Direct3D is a drawing interface for 3D hardware. It integrates tightly with DirectDraw as its buffer-management system, allowing DirectDraw surfaces to be used both as 3D rendering targets and as source texture maps. This allows for hardware-decompressed motion-video mapping, hardware 3D rendering in 2D overlay planes, or even sprites, for example.

Direct3D is designed to set a standard for hardware acceleration—including geometry transformations, 3D clipping, and lighting. Direct3D provides a highly optimized, software-only implementation of the full 3D rendering pipeline. Any part or all of this pipeline can be replaced at any stage by accelerating hardware. This allows you to write applications now that will be able to use better hardware acceleration as new hardware is developed.

Direct3D is tightly integrated with DirectDraw. The DirectDraw driver COM interfaces and the Direct3D driver COM interface both allow you to communicate with the same underlying object. For more information about the integration of Direct3D and DirectDraw, see **Direct3D Integration with DirectDraw**. For information about DirectDraw's support the 3D surfaces, see **Support for 3D Surfaces**.

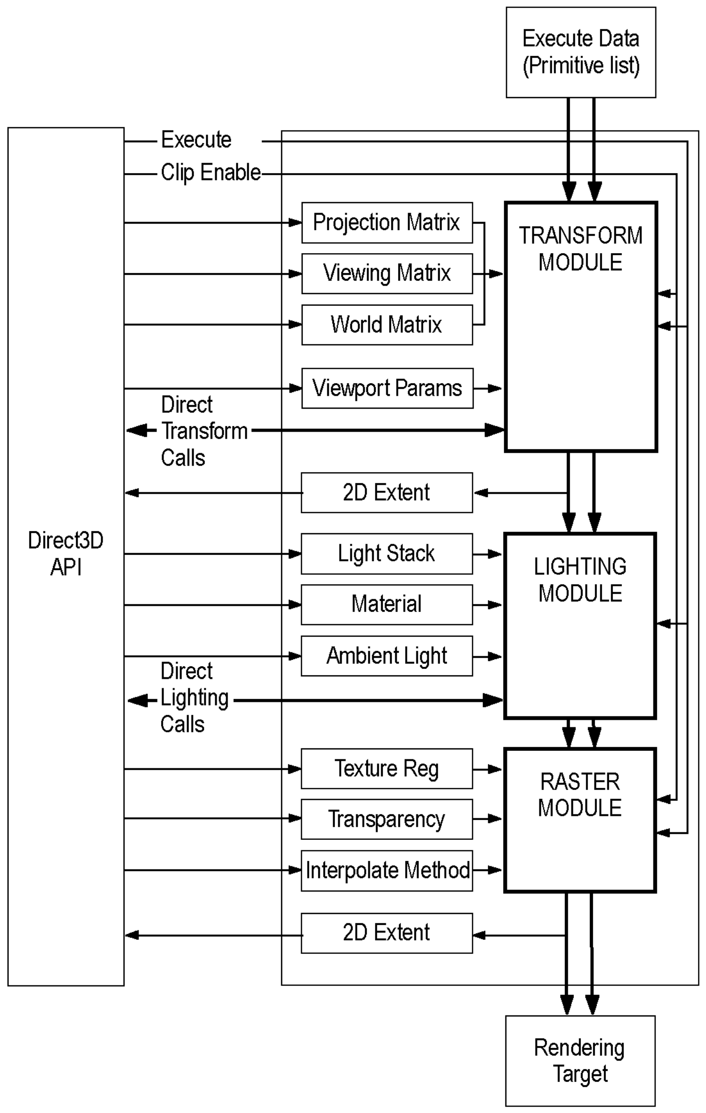
Much of the information in this section describes implementation details of Direct3D that do not directly apply to those of you who use the Retained-Mode interface. If you use the Immediate-Mode interface, however, you'll need a good understanding of these implementation details. Those of you who use the Retained-Mode interface will also benefit from a good theoretical grounding in the system architecture.

Rendering Engine

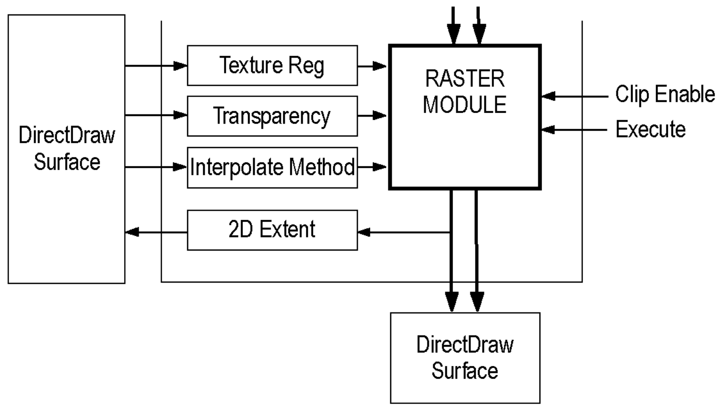
The Direct3D architecture is based on a virtual 3D rendering engine composed of three separate modules:

- The **transformation module** handles the transformation of geometry by using three 4-by-4 matrices, one each for the view, world, and projection transformations. This module supports arbitrary projection matrices, allowing perspective and orthographic views.
- The **lighting module** calculates lighting information for geometry, supporting ambient, directional, point, and spotlight light sources.
- The **rasterization module** uses the output of the geometry and lighting modules to render the scene. The scene description is in an extensible display-list–based format that can support both 2D and 3D primitives.

The following illustration shows how the three modules of the rendering engine interact with the rest of the Direct3D architecture.



The rasterization module interacts with DirectDraw as shown in the following illustration. Direct3D makes use of DirectDraw surfaces as render targets and texture sources.



Each of these modules can be hardware-accelerated or emulated in software. Direct3D can be queried to verify which components are currently running under emulation. When used together, these modules form the Direct3D rendering pipeline.

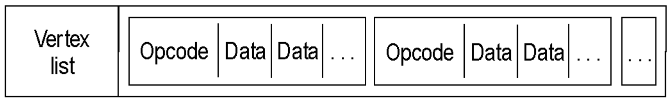
All three modules are dynamically loadable and can be changed on the fly between rendered frames. This allows new modules to be swapped in for either hardware acceleration or different rendering effects. Direct3D comes with one transformation module but a choice of two lighting and two rasterization modules. This provides greater flexibility in lighting and rendering, allowing the possibility, for example, of rendering more realistic scenes by simply switching the lighting module. An independent vendor could even provide its own special-effects-rasterization modules.

Execute Buffers

Each of the three modules in the rendering engine maintains state information that is set by using the Direct3D API. After all the state information is set, the rendering engine is ready to process display lists, which are known as *execute buffers*. Your application works explicitly with execute buffers only in Immediate Mode; Retained-Mode applications work at a higher level than this.

Execute buffers are fully self-contained, independent packets of information. They contain a vertex list followed by an instruction stream. The instruction stream consists of operation codes, or *opcodes*, and the data that is operated on by those opcodes. Direct3D's 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.

The following illustration shows the format of execute buffers.



The instructions define how the vertex list should be lit and rendered. 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.

The hardware determines the size of the execute buffer. You can retrieve this size by calling the **IDirect3DDevice::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.

You can disable the lighting module or both the lighting and transformation modules 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.

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.

Transformation Module

The transformation module has four state registers that the user can modify: the viewport, the viewing matrix, the world matrix, and the projection matrix. Whenever one of these parameters is modified, they are combined to form a new transformation matrix, which is also maintained by the transformation module. The transformation matrix defines the rotation and projection of a set of 3D vertices from their model coordinates to the 2D window.

Although the application can set the transformation matrix directly, it is not recommended. A number of matrix classifications take place in the combination phase that allow optimized transformation math to be used, but this is precluded if the application specifies the matrix directly.

A display list supports a number of different vertex types. For rasterization-only hardware, the application should use the **D3DTLVERTEX** structure, which is a transformed and lit vertex—that is, it contains screen coordinates and colors. If the hardware handles the transformations, the application should use a

D3DLVERTEX structure. This structure contains only data and a color that would be filled by software lighting. The **D3DHVERTEX** structure defines a homogeneous vertex used when the application is supplying model-coordinate data that needs clipping. If the hardware supports lighting, the application simply uses a **D3DVERTEX** structure, because this type of vertex can be transformed and lit during rendering. The software emulation driver supports all of these vertex types.

There are two types of methods for the transformation module: those that set the state of the transformation module and those that use the transformation module directly to act on a set of vertices. Calling the transformation module directly is useful for testing bounding volumes or for transforming a set of vectors. These operations transform geometry by using the current transformation matrices. They can also perform clipping tests against the current viewing volume. The structure used for all the direct transformation functions is **D3DTRANSFORMDATA**.

Lighting Module

The lighting module maintains a stack of current lights, the ambient light level, and a material.

When using the lighting module directly, each element of input data to the lighting module (the **D3DLIGHTINGELEMENT** structure) contains a direction vector and a position (for positional light sources such as point lights and spotlights).

Two lighting models—monochromatic and RGB—are supported. The color fields are always placed after the **D3DLIGHTINGELEMENT** structure in the **D3DLIGHTDATA** structure.

The monochromatic lighting model (sometimes also called the "ramp" lighting model) uses the gray component of each light to calculate a single shade value. The RGB lighting model produces more realistic results, using the full color content of light sources and materials to calculate the lit colors.

For materials with no specular component, the shade is the diffuse component of the light intensity and ranges from 0 (ambient light only) to 1 (full intensity light). For materials with a specular component, the shade combines both the specular and diffuse components of the light according to the following equation:

$$shade = \frac{3}{4}(diffuse - diffuse \times specular) + specular$$

This shade value is designed to work with precalculated ramps of colors (either in the hardware's color-lookup table or in lookup tables implemented in software). These precalculated ramps are divided into two sections. A ramp of the material's diffuse color takes up the first three-quarters of the precalculated ramp; this ranges from the ambient color to the maximum diffuse color. A ramp ranging from the maximum diffuse color to the maximum specular color of the material takes up the last quarter of the precalculated ramp. For rendering, the shade value should be scaled by the size of the ramp and used as an index to look up the color required.

A packed RGB color is defined as the following:

```
#define RGB_MAKE (red, green, blue) \
     ((red)   << 16) | \
     ((green) << 8)  | \
     (blue))
```

A packed RGBA color is defined as the following:

```
#define RGBA_MAKE(red, green, blue, alpha) \
    (((alpha) << 24) | \
     ((red)   << 16) | \
     ((green) << 8)  | \
     (blue))
```

Colors in Direct3D are defined as follows:

```
typedef unsigned long D3DCOLOR;
```

The type of the light must be one of the members of the **D3DLIGHTTYPE** enumerated type: D3DLIGHT_DIRECTIONAL, D3DLIGHT_POINT, D3DLIGHT_PARALLELPOINT, D3DLIGHT_SPOT, or D3DLIGHT_GLSPOT. This enumerated type is part of the **D3DLIGHT** structure. Another member of this structure is a **D3DCOLORVALUE** structure, which specifies the color of the light. The values given for the red, green, and blue light components typically range from 0 to 1. The value for the ramp lighting model is based on the following equation:

$$shade = 0.30red + 0.59green + 0.11blue$$

Each of the color values can fall outside of the 0 to 1 range, allowing the use of advanced lighting effects (such as dark lights). The direction vectors in the **D3DLIGHT** structure describe the direction from the model to the light source. This vector should be normalized for directional lights. All vectors should be given in world coordinates; these vectors are transformed into model coordinates by using the current world matrix, allowing the efficient lighting of models without the need to transform the vectors into world coordinates. For point lights and spotlights, the range parameter gives the effective range of the light source. Vertices that are outside this range will not be affected by the light. The intensity of the light is modified by a quadratic attenuation factor, where *d* is the distance from the vertex being lit to the light, as shown in the following equation:

$$attenuation = attenuation_0 + attenuation_1 \times d + attenuation_2 \times d^2$$

The remaining members of the **D3DLIGHT** structure (**dvTheta** and **dvPhi**) are used for spotlights to define the angles of the umbra and penumbra cones, respectively. The falloff factor (**dvFalloff**) is applied between the umbra and penumbra cones of the spotlight.

There are two types of methods for the lighting module, those that set the state of the lighting module, and those that use the lighting module directly to act on a set of points.

Like the transformation module, the lighting module can also be called directly. The **D3DLIGHTDATA** structure is used for all the direct lighting functions.

Rasterization Module

The rasterization module handles only execute calls—the calls that render execute buffers. Instructions in the execute buffer set the state for the rasterization module.

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 in view. 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.

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 0 at the front of the viewing frustum to 1 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 backfacing 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.

Colors and Fog

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

Palette Entries

Your application can use the **IDirect3DRM::CreateDeviceFromSurface** method to draw to a DirectDraw surface. 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. (You can also use the members of the **D3DRMPALETTEFLAGS** enumerated type in the **D3DRMPALETTEENTRY** structure to specify how to share palette entries.) 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 lighting mode, fog works properly only when the fog color is black or when there is no lighting, in which case any fog color has the same effect.

There are three fog modes: linear, exponential, and exponential squared. Only the linear fog mode is supported for DirectX 2.

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:

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

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 2 SDK sample code; these macros are also described in **Setting the Immediate-Mode Render State**.

Direct3D File Format

The Direct3D file format stores meshes, textures, animation sets, and user-definable objects, facilitating the exchange of 3D information between applications. Support for animation sets allows predefined paths to be stored for playback in real time. Instancing and hierarchies are also supported, allowing multiple references to a single data object (such as a mesh) while storing the data for the object only once per file.

Direct3D files have a .X file name extension. This DirectX™ 2 Software Development Kit (SDK) includes conversion tools (Conv3ds.exe and Convxof.exe) that allow you to convert files from 3DS files generated by Autodesk 3D Studio® and from XOF files that were generated for earlier versions of Direct3D.

The Direct3D file format is used natively by the Direct3D Retained-Mode API, providing support for loading predefined objects into an application and for writing mesh information constructed by the application in real time.

A Technical Foundation for 3D Programming

The following sections describe some of the technical concepts you need to understand before you write programs that incorporate 3D graphics. In these sections, you will find a general discussion of coordinate systems and transformations. This is not a discussion of broad architectural details, such as setting up models, lights, and viewing parameters. For more information about these topics, see **Introduction to Direct3D Retained-Mode Objects**.

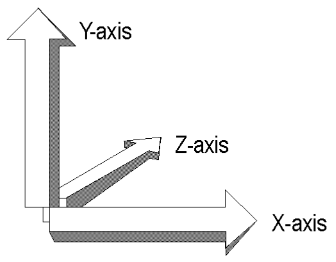
If you are already experienced in producing 3D graphics, simply scan the following sections for information that is unique to Direct3D Retained Mode.

3D Coordinate Systems

There are two varieties of Cartesian coordinate systems in 3D 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's Coordinate System

Direct3D uses the 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

Direct3D also uses *texture coordinates*. These coordinates (u and v) are used when mapping textures onto an object. The v-vector describes the direction or orientation of the texture and lies along the z-axis. The u-vector (or the *up* vector) typically lies along the y-axis, with its origin at [0,0,0]. For more information about u- and v-coordinates, see **Direct3DRMWrap**.

3D Transformations

In programs that work with 3D 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-by-4 matrix. In the following example, a matrix is used to reinterpret the point (x, y, z), producing the new point (x', y', z'):

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

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

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

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

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 the following:

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

Translation

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

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

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'):

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

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

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

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'):

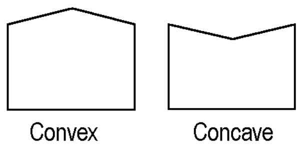
$$[x'\, y'\, z'\, 1] = [x\, y\, z\, 1]\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Polygons

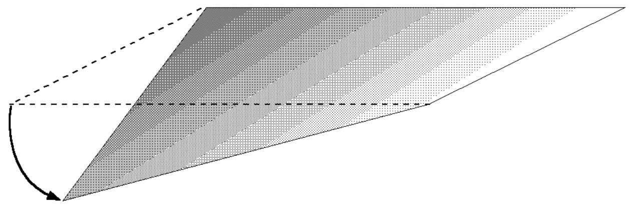
Three-dimensional objects in Direct3D are made up of meshes. A mesh is a set of faces, each of which is described by a simple polygon. The fundamental polygon type is the triangle. Although Retained-Mode applications can specify polygons with more than three vertices, the system translates these into triangles before the objects are rendered. Immediate-Mode applications must use triangles.

Geometry Requirements

Triangles are the preferred polygon type because they are always convex and they are always planar, two conditions that are required of polygons by the renderer. A polygon is convex if a line drawn between any two points of the polygon is also inside the polygon.

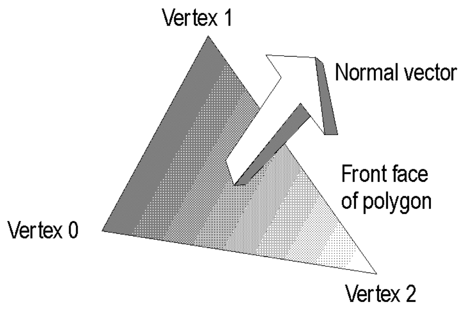


The three vertices of a triangle always describe a plane, but it is easy to accidentally create a non-planar polygon by adding another vertex.

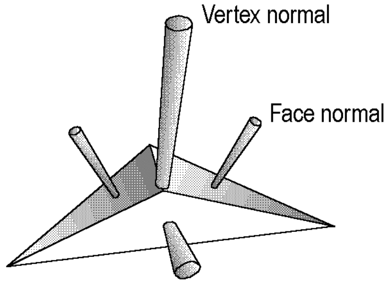


Face and Vertex Normals

Each face in a mesh has a perpendicular face normal 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 Phong and Gouraud shade modes, and for controlling lighting and texturing effects, the system uses vertex normals.

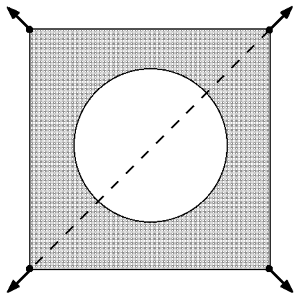


Shade Modes

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, the system calculates the appropriate shade value for each pixel on a face.

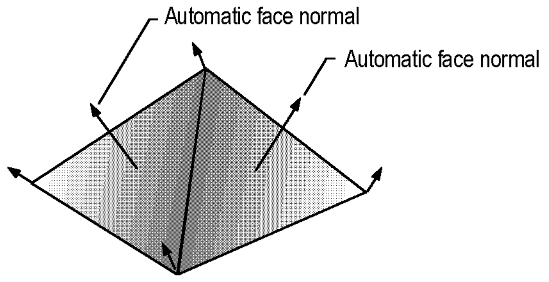
Note Phong shading is not supported for DirectX 2.

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.

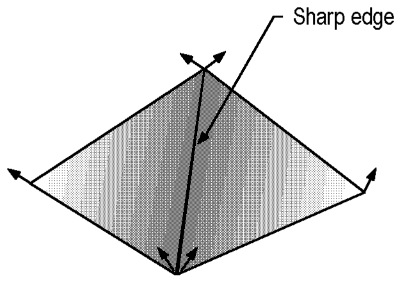


In this case, the Phong shade mode would calculate the value for each pixel and display the spotlight. The Gouraud shade mode, which interpolates between vertices, would miss the spotlight altogether; the face would be rendered as though the spotlight did not exist.

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 or Phong 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. The following 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 supported for DirectX 2.

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 model (D3DCOLOR_MONO), the system uses only the blue component of the vertex color.

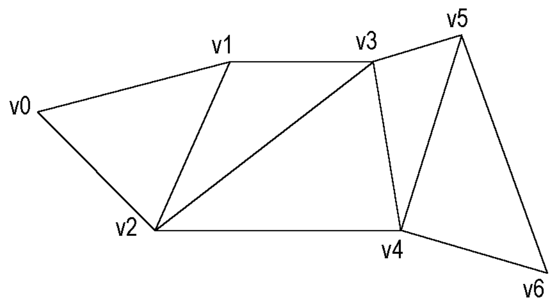
For example, if the red component of the color of vertex 1 were 0.8 and the red component of vertex 2 were 0.4, in the Gouraud shade mode and RGB color model the system would use interpolation to assign a red component of 0.6 to the pixel at the midpoint of the line between these vertices.

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.

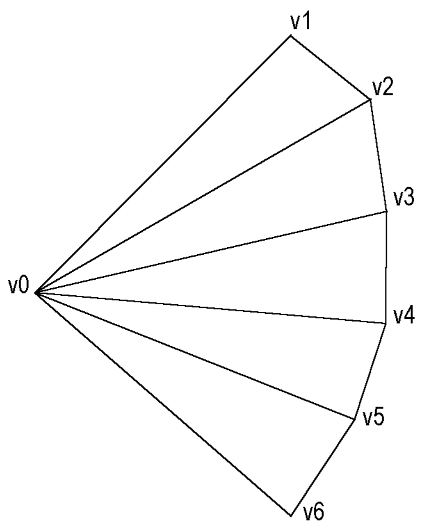
Triangle Strips and Fans

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 v0, v1, and v2 to draw the first triangle, v1, v3, and v2 to draw the second triangle, v3, v4, and v2 to draw the third, and so on. Notice that the vertices of the second triangle 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 v0, v1, and v2 to draw the first triangle, v0, v2, and v3 to draw the second triangle, and so on.

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

Vectors and Quaternions

Vectors are used throughout Direct3D to describe position and orientation. Each vertex in a primitive is described by a vector that gives its position, a normal vector that gives its orientation, texture coordinates, and a color. (In Retained Mode, the **D3DRMVERTEX** structure contains these values.)

Quaternions add a fourth element to the [*x, y, z*] values that define a vector. They are an alternative to the matrix methods that are typically used for 3D rotations. Direct3D's Retained Mode includes some functions that help you work with quaternions. For example, the **D3DRMQuaternionFromRotation** function adds a rotation value to a vector that defines an axis of rotation and returns the result in a quaternion defined by a **D3DRMQUATERNION** structure.

Retained-Mode applications can use the following functions to simplify the task of working with vectors and quaternions:

D3DRMQuaternionFromRotation

D3DRMQuaternionMultiply

D3DRMQuaternionSlerp

D3DRMVectorAdd

D3DRMVectorCrossProduct

D3DRMVectorDotProduct

D3DRMVectorModulus

D3DRMVectorNormalize

D3DRMVectorRandom

D3DRMVectorReflect

D3DRMVectorRotate

D3DRMVectorScale

D3DRMVectorSubtract

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 3D 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:

- **Clip Tests on Execution**
- **Batching Primitives**
- **Texture Size**
- **Triangle Flags**

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 Performance Tips**
- **Ramp Textures**
- **Z-Buffers**
- **Copy Mode**

Clip Tests on Execution

Your application 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's Retained Mode automatically uses these features to speed up its use of execute buffers.

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.

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 by 256 are the fastest. If your application uses four 128-by-128 textures, for example, try to ensure that they use the same palette and place them all into one 256-by-256 texture. This technique also reduces the amount of texture swapping. Of course, you should not use 256-by-256 textures unless your application requires that much

texturing because, as already mentioned, textures should be kept as small as possible.

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**.

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-by-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.

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.

Z-Buffers

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 scanline basis. If a scanline 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. This allows for fast bounding-box rejection of occluded geometry.

The Retained-Mode API can automatically order its scenes from front to back to facilitate z-buffer optimization. Retained Mode also z-tests primitives for all meshes that contain more than a few hundred triangles.

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 **IDirectDrawSurface::Lock** and **IDirectDrawSurface::Unlock** methods, as well as between calls to the **IDirectDrawSurface::GetDC** and **IDirectDrawSurface::ReleaseDC** methods.)

Copy Mode

Applications that use the ramp driver can sometimes improve performance by using the D3DTBLEND_COPY texture-blending mode from the **D3DTEXTUREBLEND** enumerated type.

To use copy mode, your application's textures must use the same pixel format as the primary surface and also must use the same palette as the primary surface. Copy mode does no lighting and simply copies texture pixels to the screen. This is often a good technique for prelit textured scenes.

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. Again, you can use D3dtest.exe to verify system performance in this case.

Direct3D Retained-Mode Overview

About Retained Mode

This section describes Direct3D's Retained Mode, Microsoft's solution for real-time 3D graphics on the personal computer. If you need to create a 3D environment and manipulate it in real time, you should use Direct3D's Retained-Mode API.

Direct3D is integrated tightly with DirectDraw. A DirectDraw object encapsulates both the DirectDraw and Direct3D states—your application can use the **IDirectDraw::QueryInterface** method to retrieve an **IDirect3D** interface to a DirectDraw object. For more information about this integration, see **Direct3D Driver Interface**.

If you have written code that uses 3D graphics before, many of the concepts underlying Retained Mode will be familiar to you. If, however, you are new to 3D programming, you should pay close attention to **Introduction to Direct3D Retained-Mode Objects**, and you should read **A Technical Foundation for 3D Programming**. Whether you are new to 3D programming or just beginning, you should look carefully at the sample code included with this SDK; it illustrates how to put Retained Mode to work in real-world applications.

This section is an introduction to 3D programming. It describes Microsoft's 3D-graphics solutions and some of the technical background you need to manipulate points in three dimensions. It is not an introduction to programming with Direct3D's Retained Mode; for this information, see **Direct3D Retained-Mode Tutorial**.

Introduction to Direct3D Retained-Mode Objects

All access to Direct3D Retained Mode is through a small set of objects. The following table lists these objects and a brief description of each:

Object	Description
Direct3DRMAnimation	This object defines how a transformation will be modified, often in reference to a Direct3DRMFrame object; therefore, you can use it to animate the position, orientation, and scaling of Direct3DRMVisual, Direct3DRMLight, and Direct3DRMViewport objects.
Direct3DRMAnimationSet	This object allows Direct3DRMAnimation objects to be grouped together.
Direct3DRMDevice	This object represents the visual display destination for the renderer.
Direct3DRMFace	This object represents a single polygon in a mesh.
Direct3DRMFrame	This object positions objects within a scene and defines the positions and

	orientations of visual objects.
Direct3DRMLight	This object defines one of five types of lights that are used to illuminate the visual objects in a scene.
Direct3DRMMaterial	This object defines how a surface reflects light.
Direct3DRMMesh	This object consists of a set of polygonal faces. You can use this object to manipulate groups of faces and vertices.
Direct3DRMMeshBuilder	This object allows you to work with individual vertices and faces in a mesh.
Direct3DRMObject	This object is a base class used by all other Direct3D Retained-Mode objects; it has characteristics that are common to all objects.
Direct3DRMPickedArray	This object identifies a visual object that corresponds to a given 2D point.
Direct3DRMShadow	This object defines a shadow.
Direct3DRMTexture	This object is a rectangular array of colored pixels.
Direct3DRMUserVisual	This object is defined by an application to provide functionality not otherwise available in the system.
Direct3DRMViewport	This object defines how the 3D scene is rendered into a 2D window.
Direct3DRMVisual	This object is anything that can be rendered in a scene. Visual objects need not be visible; for example, a frame can be added as a visual object.
Direct3DRMWrap	This object calculates texture coordinates for a face or mesh.

Many objects can be grouped into arrays, called *array objects*. Array objects make it simpler to apply operations to the entire group. The COM interfaces that allow you to work with array objects contain the **GetElement** and **GetSize** methods. These methods retrieve a pointer to an element in the array and the size of the array, respectively. For more information about array interfaces, see **Introduction to Array Interfaces**.

Objects and Interfaces

Calling the *IObjectName*::**QueryInterface** method retrieves a valid interface pointer only if the object supports that interface; therefore, you could call the **IDirect3DRMDevice::QueryInterface** method to retrieve the

IDirect3DRMWinDevice interface, but not to retrieve the **IDirect3DRMVisual** interface.

Object name	**Supported interfaces**
Direct3DRMAnimation	**IDirect3DRMAnimation**
Direct3DRMAnimationSet	**IDirect3DRMAnimationSet**
Direct3DRMDevice	**IDirect3DRMDevice**, **IDirect3DRMWinDevice**
Direct3DRMFace	**IDirect3DRMFace**
Direct3DRMFrame	**IDirect3DRMFrame**, **IDirect3DRMVisual**
Direct3DRMLight	**IDirect3DRMLight**
Direct3DRMMaterial	**IDirect3DRMMaterial**
Direct3DRMMesh	**IDirect3DRMMesh**, **IDirect3DRMVisual**
Direct3DRMMeshBuilder	**IDirect3DRMMeshBuilder**, **IDirect3DRMVisual**
Direct3DRMShadow	**IDirect3DRMShadow**, **IDirect3DRMVisual**
Direct3DRMTexture	**IDirect3DRMTexture**, **IDirect3DRMVisual**
Direct3DRMUserVisual	**IDirect3DRMUserVisual**, **IDirect3DRMVisual**
Direct3DRMViewport	**IDirect3DRMViewport**
Direct3DRMWrap	**IDirect3DRMWrap**

The following code sample illustrates how to create two interfaces to a single Direct3DRMDevice object. The **IDirect3DRM::CreateObject** method creates an uninitialized Direct3DRMDevice object. The **IDirect3DRMDevice::InitFromClipper** method initializes the object. The call to the **IDirect3DRMDevice::QueryInterface** method creates a second interface to the Direct3DRMDevice object—an **IDirect3DRMWinDevice** interface the application will use to handle WM_PAINT and WM_ACTIVATE messages.

```
d3drmapi->CreateObject(CLSID_CDirect3DRMDevice, NULL,
    IID_IDirect3DRMDevice,(LPVOID FAR*)&dev1);
dev1->InitFromClipper(lpDDClipper, IID_IDirect3DRMDevice,
    r.right, r.bottom);
dev1->QueryInterface(IID_IDirect3DRMWinDevice, (LPVOID*) &dev2);
```

To determine if two interfaces refer to the same object, call the **QueryInterface** method of each interface and compare the values of the pointers they return. If the pointer values are the same, the interfaces refer to the same object.

All Direct3D Retained-Mode objects support the **IDirect3DRMObject** and **IUnknown** interfaces in addition to the interfaces in the preceding list. Array objects are not derived from **IDirect3DRMObject**, however. Array objects have no class identifiers (CLSIDs) because they are not needed. Applications cannot create array objects in a call to the **IDirect3DRM::CreateObject** method; instead, they should use the creation methods listed below for each interface:

Array interface	**Creation method**
IDirect3DRMDeviceArray	**IDirect3DRM::GetDevices**
IDirect3DRMFaceArray	**IDirect3DRMMeshBuilder::GetFaces**
IDirect3DRMFrameArray	**IDirect3DRMPickedArray::GetPick** or **IDirect3DRMFrame::GetChildren**
IDirect3DRMLightArray	**IDirect3DRMFrame::GetLights**
IDirect3DRMPickedArray	**IDirect3DRMViewport::Pick**
IDirect3DRMViewportArray	**IDirect3DRM::CreateFrame**
IDirect3DRMVisualArray	**IDirect3DRMFrame::GetVisuals**

Objects and Reference Counting

Whenever an object is created, its reference count is increased. Each time an application creates a child of an object or a method returns a pointer to an object, the system increases the reference count for that object. The object is not deleted until its reference count reaches zero.

Applications should need to keep track of the reference count for a single object only: the root of the scene. The system keeps track of the other reference counts automatically. Applications should be able to simply release the scene, the viewport, and the device when they clean up before exiting. (When your application releases the viewport, the system automatically takes care of the camera's references.) An application could theoretically release a viewport without releasing the device, such as if it needed to add a new viewport to the device, but whenever an application releases a device, it should release the viewport as well.

The reference count of a frame object increases whenever a child frame or visual object is added to the frame. When you use the **IDirect3DRMFrame::AddChild** method to move a child from one parent to another, the system handles the reference counting automatically.

After your application loads a visual object into a scene, the scene handles the reference counting for the visual object. The application no longer needs the visual object and can release it.

Creating and applying a wrap does not increase the reference count of any objects, because wrapping is really just a convenient method of calculating texture coordinates.

Direct3DRMAnimation and Direct3DRMAnimationSet

An animation in Retained Mode is defined by a set of *keys*. A key is a time value associated with a scaling operation, an orientation, or a position. A Direct3DRMAnimation object defines how a transformation is modified according to the time value. The animation can be set to operate on a Direct3DRMFrame object, so it could be used to animate the position, orientation, and scaling of Direct3DRMVisual, Direct3DRMLight, and Direct3DRMViewport objects.

The**IDirect3DRMAnimation::AddPositionKey**, **IDirect3DRMAnimation::AddRotateKey**, and **IDirect3DRMAnimation::AddScaleKey** methods each specify a time value whose units are arbitrary. If an application adds a position key with a time value of 99, for example, a new position key with a time value of 49 would occur exactly halfway between the (zero-based) beginning of the animation and the first position key.

The animation is driven by calling the **IDirect3DRMAnimation::SetTime** method. This sets the visual object's transformation to the interpolated position, orientation, and scale of the nearby keys in the animation. As with the methods that add animation keys, the time value for **IDirect3DRMAnimation::SetTime** is an arbitrary value, based on the positions of keys the application has already added.

A Direct3DRMAnimationSet object allows Direct3DRMAnimation objects to be grouped together. This allows all the animations in an animation set to share the same time parameter, simplifying the playback of complex articulated animation sequences. An application can add an animation to an animation set by using the **IDirect3DRMAnimationSet::AddAnimation** method, and it can remove one by using the **IDirect3DRMAnimationSet::DeleteAnimation** method. Animation sets are driven by calling the **IDirect3DRMAnimationSet::SetTime** method.

For related information, see the **IDirect3DRMAnimation** and **IDirect3DRMAnimationSet** interfaces.

Direct3DRMDevice and Direct3DRMDeviceArray

All forms of rendered output must be associated with an output device. The device object represents the visual display destination for the renderer.

The renderer's behavior depends on the type of output device that is specified. You can define multiple viewports on a device, allowing different aspects of the scene to be viewed simultaneously. You can also specify any number of devices, allowing multiple destination devices for the same scene.

Retained Mode supports devices that render directly to the screen, to windows, or into application memory.

For related information, see the **IDirect3DRMDevice** interface.

Quality

The device allows the scene and its component parts to be rendered with various degrees of realism. Each mesh can have its own quality, but the maximum quality available for a mesh is that of the device.

An application can change the rendering quality of a device by using the **IDirect3DRMDevice::SetQuality** and **IDirect3DRMMeshBuilder::SetQuality** methods. To retrieve the rendering quality for a device, it can use the **IDirect3DRMDevice::GetQuality** and **IDirect3DRMMeshBuilder::GetQuality** methods.

Color Models

Retained Mode supports two color models: an RGB model and a monochromatic (or ramp) model. To retrieve the color model, an application can use the **IDirect3DRMDevice::GetColorModel** method.

The RGB model treats color as a combination of red, green, and blue light, and it supports multiple light sources that can be colored. There is no limit to the number of colors in the scene. You can use this model with 8-, 16-, 24-, and 32-bit displays. If the display depth is less than 24 bits, the limited color resolution can produce banding artifacts; you can avoid these artifacts by using optional dithering.

The monochromatic model also supports multiple light sources, but their color content is ignored. Each source is set to a gray intensity. RGB colors at a vertex are interpreted as brightness levels, which (in Gouraud shading) are interpolated across a face between vertices with different brightnesses. The number of differently colored objects in the scene is limited; after all the system's free palette entries are used up, the system's internal palette manager finds colors that already exist in the palette and that most closely match the intended colors. Like the RGB model, you can use this model with 8-, 16-, 24-, and 32-bit displays. (The monochromatic model supports only 8-bit textures, however.) The advantage of the monochromatic model over the RGB model is simply performance.

It is not possible to change the color model of a Direct3D device. Your application should use the **IDirect3D::EnumDevices** or **IDirect3D::FindDevice** method to identify a driver that supports the required color model, then specify this driver in one of the device-creation methods.

Window Management

For correct operation, applications must inform Retained Mode when WM_MOVE, WM_PAINT, and WM_ACTIVATE messages are received from the operating system by using the **IDirect3DRMWinDevice::HandlePaint** and **IDirect3DRMWinDevice::HandleActivate** methods.

For related information, see **IDirect3DRMWinDevice**.

Direct3DRMFace and Direct3DRMFaceArray

A face represents a single polygon in a mesh. An application can set the color, texture, and material of the face by using the **IDirect3DRMFace::SetColor**, **IDirect3DRMFace::SetColorRGB**, **IDirect3DRMFace::SetTexture**, and **IDirect3DRMFace::SetMaterial** methods.

Faces are constructed from vertices by using the **IDirect3DRMFace::AddVertex** and **IDirect3DRMFace::AddVertexAndNormalIndexed** methods. An application can read the vertices of a face by using the **IDirect3DRMFace::GetVertices** and **IDirect3DRMFace::GetVertex** methods.

For related information, see **IDirect3DRMFace**.

Direct3DRMFrame and Direct3DRMFrameArray

The term *frame* is derived from an object's physical frame of reference. The frame's role in Retained Mode is similar to a window's role in a windowing system. Objects can be placed in a scene by stating their spatial relationship to a relevant reference frame; they are not simply placed in world space. A frame is used to position objects in a scene, and visuals take their positions and orientation from frames.

A *scene* in Retained Mode is defined by a frame that has no parent frame; that is, a frame at the top of the hierarchy of frames. This frame is also sometimes called a *root frame* or *master frame*. The scene defines the frame of reference for all of the other objects. You can create a scene by calling the **IDirect3DRM::CreateFrame** function and specifying NULL for the first parameter.

You should be comfortable with Direct3D's left-handed coordinate system before working with frames. For more information about coordinate systems, see **3D Coordinate Systems**.

Hierarchies

The frames in a scene are arranged in a tree structure. Frames can have a parent frame and child frames. Remember, a frame that has no parent frame defines a scene.

Child frames have positions and orientations relative to their parent frames. If the parent frame moves, the child frames also move.

An application can set the position and orientation of a frame relative to any other frame in the scene, including the root frame if it needs to set an absolute position.

You can also remove frames from one parent frame and add them to another at any time by using the **IDirect3DRMFrame::AddChild** method. To remove a child frame entirely, use the **IDirect3DRMFrame::DeleteChild** method. To retrieve a frame's child and parent frames, use the **IDirect3DRMFrame::GetChildren** and **IDirect3DRMFrame::GetParent** methods.

You can add frames as visuals to other frames, allowing you to use a given hierarchy many times throughout a scene. The new hierarchies are referred to as *instances*. Be careful to avoid instancing a parent frame into its children, because that will degrade performance. Retained Mode does no run-time checking for cyclic hierarchies. You cannot create a cyclic hierarchy by using the methods of the **IDirect3DRMFrame** interface; instead, this is possible only when you add a frame as a visual.

Transformations

You can also think of the position and orientation of a frame relative to its parent frame as a linear transformation that takes vectors defined relative to the child frame and changes them to equivalent vectors defined relative to the parent.

Transformations can be represented by 4-by-4 matrices, and coordinates can be represented by four-element row vectors, [*x, y, z,* 1].

If v_{child} is a coordinate in the child frame, then v_{parent}, the equivalent coordinate in the parent frame, is defined as:

$$v_{\text{parent}}=v_{\text{child}}T_{\text{child}}$$

T_{child} is the child frame's transformation matrix.

The transformations of all the parent frames above a child frame up to the root frame are concatenated with the transformation of that child to produce a world transformation. This world transformation is then applied to the visuals on the child frame before rendering. Coordinates relative to the child frame are sometimes called *model coordinates*. After the world transformation is applied, coordinates are called *world coordinates*.

The transformation of a frame can be modified directly by using the **IDirect3DRMFrame::AddTransform**, **IDirect3DRMFrame::AddScale**, **IDirect3DRMFrame::AddRotation**, and **IDirect3DRMFrame::AddTranslation** methods. Each of these methods specifies a member of the **D3DRMCOMBINETYPE** enumerated type, which specifies how the matrix supplied by the application should be combined with the current frame's matrix.

The **IDirect3DRMFrame::GetRotation** and **IDirect3DRMFrame::GetTransform** methods allow you to retrieve a frame's rotation axis and transformation matrix. To change the rotation of a frame, use the **IDirect3DRMFrame::SetRotation** method.

Use the **IDirect3DRMFrame::Transform** and **IDirect3DRMFrame::InverseTransform** methods to change between world coordinates and frame coordinates.

Motion

Every frame has an intrinsic rotation and velocity. Frames that are neither rotating nor translating simply have zero values for these attributes. These attributes are used before each scene is rendered to move objects in the scene, and they can also be used to create simple animations.

Callback Functions

Frames support a callback function that you can use to support more complex animations. The application registers a function that the frame calls before the motion attributes are applied. Where there are multiple frames in a hierarchy, each with associated callback functions, the parent frames are called before the child frames. For a given hierarchy, rendering does not take place until all of the required callback functions have been invoked.

To add this callback function, use the **IDirect3DRMFrame::AddMoveCallback** method; to remove it, use the **IDirect3DRMFrame::DeleteMoveCallback** method.

You can use these callback functions to provide new positions and orientations from a preprogrammed animation sequence or to implement dynamic motion in which the activities of visuals depend upon the positions of other objects in the scene.

Direct3DRMLight and Direct3DRMLightArray

Lighting effects are employed to increase the visual fidelity of a scene. The system colors each object based on the object's orientation to the light sources in the scene. The contribution of each light source is combined to determine the color of the object during rendering. All lights have color and intensity that can be varied independently.

An application can attach lights to a frame to represent a light source in a scene. When a light is attached to a frame, it illuminates visual objects in the scene. The frame provides both position and orientation for the light. In other words, the light originates from the origin of the frame it is attached to. An application can move and redirect a light source simply by moving and reorienting the frame the light source is attached to.

Each viewport owns one or more lights. No light can be owned by more than one viewport. For more information about the interdependencies of Direct3D components, see **Object Connectivity**.

Retained Mode currently provides five types of light sources: ambient, directional, parallel point, point, and spotlight.

Ambient

An *ambient* light source illuminates everything in the scene, regardless of the orientation, position, and surface characteristics of the objects in the scene. Because 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.

Directional

A *directional* light source has orientation but no position. The light 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. The directional source 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 *parallel point* light source illuminates objects with parallel light, but the orientation of the light is taken from the position of the parallel point light source. That is, like a directional light source, a parallel point light source has orientation, but it also has position. 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.

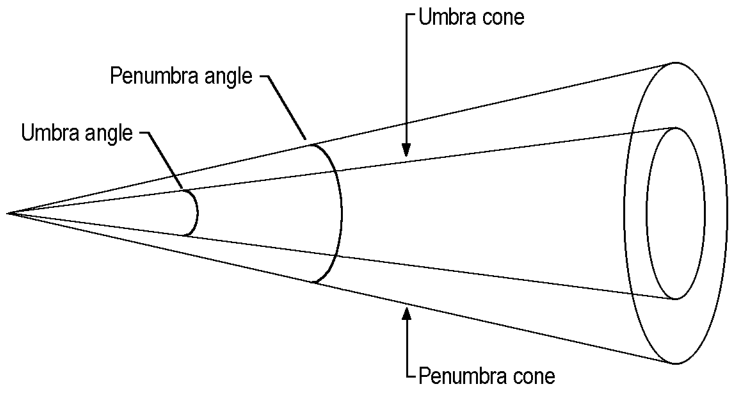
Point

A *point* light source radiates light equally in all directions from its origin. It requires the calculation of a new lighting vector for every facet or normal it illuminates, and for this reason it is computationally more expensive than a parallel point light source. It does, however, produce a more faithful lighting effect and should be chosen where visual fidelity is the deciding concern.

Spotlight

A *spotlight* 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. The angles of these two sections can be individually specified by using the **IDirect3DRMLight::GetPenumbra**, **IDirect3DRMLight::GetUmbra**,

IDirect3DRMLight::SetPenumbra, and **IDirect3DRMLight::SetUmbra** methods.



Direct3DRMMaterial

A material defines how a surface reflects light. A material has two components: an emissive property (whether it emits light) and a specular property, whose brightness is determined by a power setting. The value of the power determines the sharpness of the reflected highlights, with a value of 5 giving a metallic appearance and higher values giving a more plastic appearance.

An application can control the emission of a material by using the **IDirect3DRMMaterial::GetEmissive** and **IDirect3DRMMaterial::SetEmissive** methods, the specular component by using the **IDirect3DRMMaterial::GetSpecular** and **IDirect3DRMMaterial::SetSpecular** methods, and the power by using the **IDirect3DRMMaterial::GetPower** and **IDirect3DRMMaterial::SetPower** methods.

Direct3DRMMesh and Direct3DRMMeshBuilder

A mesh is a visual object that is made up of a set of polygonal faces. A mesh defines a set of vertices and a set of faces (the faces are defined in terms of the vertices and normals of the mesh). Changing a vertex or normal that is used by several faces changes the appearance of all faces sharing it.

The vertices of a mesh define the positions of faces in the mesh, and they can also be used to define 2D coordinates within a texture map.

You can manipulate meshes in Retained Mode by using two COM interfaces: **IDirect3DRMMesh** and **IDirect3DRMMeshBuilder**. **IDirect3DRMMesh** is very fast, and you should use it when a mesh is subject to frequent changes, such as when morphing. **IDirect3DRMMeshBuilder** is built on top of the **IDirect3DRMMesh** interface. Although the **IDirect3DRMMeshBuilder** interface is a convenient way to perform operations on individual faces and vertices, the system must convert a Direct3DRMMeshBuilder object into a Direct3DRMMesh object before rendering it. For meshes that do not change or that change infrequently, this conversion has a negligible impact on performance.

If an application needs to assign the same characteristics (such as material or texture) to several vertices or faces, it can use the **IDirect3DRMMesh** interface to combine them in a group. If the application needs to share vertices between two different groups (for example, if neighboring faces in a mesh are different colors), the vertices must be duplicated in both groups. The **IDirect3DRMMesh::AddGroup** method assigns a group identifier to a collection of faces. This identifier is used to refer to the group in subsequent calls.

The **IDirect3DRMMeshBuilder** and **IDirect3DRMMesh** interfaces allow an application to create faces with more than three sides. They also automatically split a mesh into multiple buffers if, for example, the hardware the application is rendering to has a limit of 64K and a mesh is larger than that size. These features set the Direct3DRMMesh and Direct3DRMMeshBuilder API apart from the Direct3D API.

You can add vertices and faces individually to a mesh by using the **IDirect3DRMMeshBuilder::AddVertex**, **IDirect3DRMMeshBuilder::AddFace**, and **IDirect3DRMMeshBuilder::AddFaces** methods.

You can define individual color, texture, and material properties for each face in the mesh, or for all faces in the mesh at once, by using the **IDirect3DRMMesh::SetGroupColor**, **IDirect3DRMMesh::SetGroupColorRGB**, **IDirect3DRMMesh::SetGroupTexture**, and **IDirect3DRMMesh::SetGroupMaterial** methods.

For a mesh to be rendered, you must first add it to a frame by using the **IDirect3DRMFrame::AddVisual** method. You can add a single mesh to multiple frames to create multiple instances of that mesh.

Your application can use flat, Gouraud, and Phong shade modes, as specified by a call to the **IDirect3DRMMesh::SetGroupQuality** method. (Phong shading is not available for DirectX 2, however.) This method uses values from the **D3DRMRENDERQUALITY** enumerated type. For more information about shade modes, see **Polygons**.

You can set normals (which should be unit vectors), or normals can be calculated by averaging the face normals of the surrounding faces by using the **IDirect3DRMMeshBuilder::GenerateNormals** method.

Direct3DRMObject

Direct3DRMObject is the common superclass of all objects in the system. A Direct3DRMObject object has characteristics common to all objects.

Each Direct3DRMObject object is instantiated as a COM object. In addition to the methods of the **IUnknown** interface, each object has a standard set of methods that are generic to all.

To create an object, the application must first have instantiated a Direct3D Retained-Mode object by calling the **Direct3DRMCreate** function. The application then calls the method of the object's interface that creates an object, and it specifies parameters specific to the object. For example, to create a Direct3DRMAnimation object, the application would call the **IDirect3DRM::CreateAnimation** method. The creation method then creates a new object, initializes some of the object's attributes from data passed in the parameters (leaving all others with their default values), and returns the object. Applications can then specify the interface for this object to modify and use the object.

Any object can store 32 bits of application-specific data. This data is not interpreted or altered by Retained Mode. The application can read this data by using the **IDirect3DRMObject::GetAppData** method, and it can write to it by using the **IDirect3DRMObject::SetAppData** method. Finding this data is simpler if the application keeps a structure for each Direct3DRMFrame object. For example, if calling the **IDirect3DRMFrame::GetParent** method retrieves a Direct3DRMFrame object, the application can easily retrieve the data by using a pointer to its private structure, possibly avoiding a time-consuming search.

You might also want to assign a name to an object to help you organize an application or as part of your application's user interface. You can use the **IDirect3DRMObject::SetName** and **IDirect3DRMObject::GetName** methods to set and retrieve object names.

Another example of possible uses for application-specific data is when an application needs to group the faces within a mesh into subsets (for example, for front and back faces). You could use the application data in the face to note in which of these groups a face should be included.

An application can specify a function to call when an object is destroyed, such as when the application needs to deallocate memory associated with the object. To do this, use the **IDirect3DRMObject::AddDestroyCallback** method. To remove a function previously registered with this method, use the **IDirect3DRMObject::DeleteDestroyCallback** method.

The callback function is called only when the object is destroyed—that is, when the object's reference count has reached 0 and the system is about to deallocate the memory for the object. If an application kept additional data about an object (so that its dynamics could be implemented, for example), the application could use this callback function as a way to notify itself that it can dispose of the data.

For related information, see **IDirect3DRMObject**.

Direct3DRMPickedArray

Picking is the process of searching for visuals in a scene, given a 2D coordinate in a viewport. You can use the **IDirect3DRMViewport::Pick** method to retrieve an **IDirect3DRMPickedArray** interface, and then call the **IDirect3DRMPickedArray::GetPick** method to retrieve an **IDirect3DRMFrameArray** interface and a visual object. The array of frames is the path through the hierarchy leading to the visual object; that is, a hierarchical list of the visual object's parent frames, with the top-most parent in the hierarchy first in the array.

Direct3DRMShadow

Applications can produce an initialized and usable shadow simply by calling the **IDirect3DRM::CreateShadow** method. The **IDirect3DRMShadow** interface exists so that applications that create a shadow by using the **IDirect3DRM::CreateObject** method can initialize the shadow by calling the **IDirect3DRMShadow::Init** method.

Direct3DRMTexture

A texture is a rectangular array of colored pixels. (The rectangle does not necessarily have to be square, although 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.

An **IDirect3DRMTexture** interface is actually an interface to a DirectDrawSurface object, not to a distinct Direct3D texture object. For more information about the relationship between textures in Direct3D and surfaces in DirectDraw, see **Direct3D Texture Interface**.

Your application can use the **IDirect3DRM::CreateTexture** method to create a texture from a **D3DRMIMAGE** structure, or the **IDirect3DRM::CreateTextureFromSurface** method to create a texture from a DirectDraw surface. The **IDirect3DRM::LoadTexture** method allows your application to load a texture from a file; the texture should be in Windows bitmap (.bmp) or Portable Pixmap (.ppm) format.

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. For more information, see **Direct3DRMWrap**.

Decals

Textures can also be rendered directly, as visuals. Textures used this way are sometimes known as *decals*, a term adopted by Retained Mode. A decal is rendered into a viewport-aligned rectangle. The rectangle can optionally be scaled by the depth component of the decal's position. The size of the decal is taken from a rectangle defined relative to the containing frame by using the **IDirect3DRMTexture::SetDecalSize** method. (An application can retrieve the size of the decal by using the **IDirect3DRMTexture::GetDecalSize** method.) The decal is then transformed and perspective projection is applied.

Decals have origins that your application can set and retrieve by using the **IDirect3DRMTexture::SetDecalOrigin** and **IDirect3DRMTexture::GetDecalOrigin** methods. The origin is an offset from the top-left corner of the decal. The default origin is [0, 0]. The decal's origin is aligned with its frame's position when rendering.

Texture Colors

You can set and retrieve the number of colors that are used to render a texture by using the **IDirect3DRMTexture::SetColors** and **IDirect3DRMTexture::GetColors** methods.

If your application uses the RGB color model, you can use 8-bit, 24-bit, and 32-bit textures. If you use the monochromatic (or ramp) color model, however, you can use only 8-bit textures.

Several shades of each color are used when lighting a scene. An application can set and retrieve the number of shades used for each color by calling the **IDirect3DRMTexture::SetShades** and **IDirect3DRMTexture::GetShades** methods.

A Direct3DRMTexture object uses a **D3DRMIMAGE** structure to define the bitmap that the texture will be rendered from. If the application provides the **D3DRMIMAGE** structure, the texture can easily be animated or altered during rendering.

Mipmaps

A mipmap is a sequence of textures, each of which is a progressively lower resolution, prefiltered representation of the same image. Mipmapping is a computationally low-cost way of improving the quality of rendered textures. Each prefiltered image, or level, in the mipmap is a power of two smaller than the previous level. You can specify mipmaps when filtering textures by calling the **IDirect3DRMDevice::SetTextureQuality** method.

For more information about how to use DirectDraw to create mipmaps, see **Mipmaps**.

Texture Filtering

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 **IDirect3DRMDevice::SetTextureQuality** method and the **D3DRMTEXTUREQUALITY** enumerated type to specify the texture filtering mode for your application.

Texture Transparency

You can use the **IDirect3DRMTexture::SetDecalTransparency** method to produce transparent textures. Another method for achieving transparency is to use DirectDraw's 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 that these colors should always be overwritten or never be overwritten.

For more information about DirectDraw's support for color keys, see **Color Keying**.

For related information, see **IDirect3DRMTexture**.

Direct3DRMUserVisual

User-visual objects are application-defined data that an application can add to a scene and then render, typically by using a customized rendering module. For example, an application could add sound as a user-visual object in a scene, and then render the sound during playback.

You can use the **IDirect3DRM::CreateUserVisual** method to create a user-visual object.

Direct3DRMViewport and Direct3DRMViewportArray

The viewport defines how the 3D scene is rendered into a 2D window. The viewport defines a rectangular area on a device that objects will be rendered into.

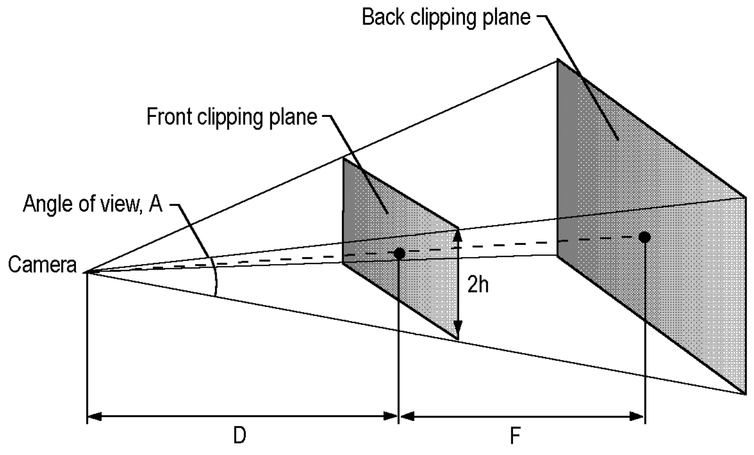
Camera

The viewport uses a Direct3DRMFrame object as a *camera*. The camera frame defines which scene is rendered and the viewing position and direction. The viewport renders only what is visible along the positive z-axis of the camera frame, with the up direction being in the direction of the positive y-axis.

An application can call the **IDirect3DRMViewport::SetCamera** method to set a camera for a given viewport. This method sets a viewport's position, direction, and orientation to that of the given camera frame. To retrieve the current camera settings, call the **IDirect3DRMViewport::GetCamera** method.

Viewing Frustum

The viewing frustum is a 3D volume in a scene positioned relative to the viewport's camera. Objects within the viewing frustum are visible. For perspective viewing, the viewing frustum is the volume of an imaginary pyramid that is between the front clipping plane and the back clipping plane.



The camera is at the tip of the pyramid, and its z-axis runs from the tip of the pyramid to the center of the pyramid's base. The front clipping plane is a distance *D* from the camera, and the back clipping plane is a distance *F* from the front clipping plane. An application can set and retrieve these values by using the **IDirect3DRMViewport::SetFront**, **IDirect3DRMViewport::SetBack**, **IDirect3DRMViewport::GetFront**, and **IDirect3DRMViewport::GetBack** methods. The height of the front clipping plane is 2*h* and defines the field of view. An application can set and retrieve the value of *h* by using the **IDirect3DRMViewport::SetField** and **IDirect3DRMViewport::GetField** methods.

The angle of view, *A*, is defined by the following equation, which can be used to calculate a value for *h* when a particular camera angle is desired:

$$A = 2 \tan^{-1} \frac{h}{D}$$

The viewing frustum is a pyramid only for perspective viewing. For orthographic viewing, the viewing frustum is cuboid. These viewing types (or projection types) are defined by the **D3DRMPROJECTIONTYPE** enumerated type and used by the **IDirect3DRMViewport::GetProjection** and **IDirect3DRMViewport::SetProjection** methods.

Transformations

To render objects with 3D coordinates in a 2D window, the object must be transformed into the camera's frame. A projection matrix is then used to give a four-element homogeneous coordinate [*x y z w*], which is used to derive a three-element coordinate [*x/w y/w z/w*], where [*x/w y/w*] is the coordinate to be used in the window and *z/w* is the depth, ranging from 0 at the front clipping plane to 1 at the back clipping plane. The projection matrix is a combination of a perspective transformation followed by a scaling and translation to scale the objects into the window.

The following matrix is the projection matrix. In these formulas, *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:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{hF}{D(F\text{-}D)} & \frac{h}{D} \\ 0 & 0 & \frac{-hF}{F\text{-}D} & 0 \end{bmatrix}$$

The following matrix is the window-scaling matrix. (The scales are dependent on the size and position of the window.) In these formulas, *s* is the window-scaling factor and *o* is the window origin:

$$W= \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & -sy & 0 & 0 \\ 0 & 0 & 1 & 0 \\ ox & oy & 0 & 1 \end{bmatrix}$$

The following matrix is the viewing matrix. This is a combination of the projection matrix and window matrix, or the dot product of P and W:

$$V = PW = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & -sy & 0 & 0 \\ \frac{ho_x}{D} & \frac{ho_y}{D} & \frac{hF}{D(F-D)} & \frac{h}{D} \\ 0 & 0 & \frac{-hF}{F-D} & 0 \end{bmatrix}$$

The scaling factors and origin sx, sy, ox, and oy are chosen so that the region [$-h$ $-h$ D] to [h h D] fits exactly into either the window's height or the window's width, whichever is larger.

The application can use the **IDirect3DRMViewport::Transform** and **IDirect3DRMViewport::InverseTransform** methods to transform vectors to screen coordinates from world coordinates and vice versa. An application can use these methods to support dragging, as shown in the following example:

```
/*
 * Drag a frame by [delta_x delta_y] pixels in the view.
 */
void DragFrame(LPDIRECT3DRMVIEWPORT view,
    LPDIRECT3DRMFRAME frame,
    LPDIRECT3DRMFRAME scene,
    int delta_x, int delta_y)
{
    D3DVECTOR p1;
    D3DRMVECTOR4D p2;

    frame->GetPosition(scene, &p1);
    view->Transform(&p2, &p1);
    p2.x += delta_x * p2.w;
    p2.y += delta_y * p2.w;
    view->InverseTransform(&p1, &p2);
    frame->SetPosition(scene, p1.x, p1.y, p1.z);
}
```

An application uses the viewport transformation to ensure that the distance by which the object is moved in world coordinates is scaled by the object's distance from the camera to account for perspective. Note that the result from **IDirect3DRMViewport::Transform** is a four-element homogeneous vector. This avoids the problems associated with coordinates being scaled by an infinite amount near the camera's position.

The viewport projection matrix produces a well-defined 3D coordinate only for points inside the viewing frustum. For a homogeneous point [x y z w] after projection, this is true if the following equations hold:

$$
\begin{aligned}
wx_{min} &\le x < wx_{max} \\
wy_{min} &\le y < wy_{max} \\
0 &\le z < w \\
&where \\
xmin &= viewport_x - viewport_{width} / 2 \\
xmax &= viewport_x + viewport_{width} / 2 \\
ymin &= viewport_y - viewport_{height} / 2 \\
ymax &= viewport_y + viewport_{height} / 2
\end{aligned}
$$

Picking

Picking is the process of searching for visuals in the scene given a 2D coordinate in the viewport's window. An application can use the **IDirect3DRMViewport::Pick** method to retrieve either the closest object in the scene or a depth-sorted list of objects.

Direct3DRMVisual and Direct3DRMVisualArray

Visuals are objects that can be rendered in a scene. Visuals are visible only when they are added to a frame in that scene. An application can add a visual to a frame by using the **IDirect3DRMFrame::AddVisual** method. The frame provides the visual with position and orientation for rendering.

You should use the **IDirect3DRMVisualArray** interface to work with groups of visual objects; although there is a **IDirect3DRMVisual** COM interface, it has no methods.

The most common visual types are Direct3DRMMeshBuilder and Direct3DRMTexture objects.

Direct3DRMWrap

You can use a wrap to calculate texture coordinates for a face or mesh. To create a wrap, the application must specify a type, a reference frame, an origin, a direction vector, and an up vector. The application must also specify a pair of scaling factors and an origin for the texture coordinates.

Your application calls the **IDirect3DRM::CreateWrap** function to create an **IDirect3DRMWrap** interface. This interface has two unique methods: **IDirect3DRMWrap::Apply**, which applies a wrap to the vertices of the object, and **IDirect3DRMWrap::ApplyRelative**, which transforms the vertices of a wrap as it is applied.

The **D3DRMMAPPING** type includes the D3DRMMAP_WRAPU and D3DRMMAP_WRAPV flags. These flags determine how the rasterizer interprets 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 D3DRMMAP_WRAPU or D3DRMMAP_WRAPV is set, the texture is a cylinder with an infinite length and 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 D3DRMMAP_WRAPU is set, the shortest line between (0.1, 0.1) and (0.9, 0.9) passes through (0, 0.5).
- If both D3DRMMAP_WRAPU and D3DRMMAP_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; applications do not set a wrap flag when more than half of a texture is applied to a single face.

Wrapping Types

There are four wrapping types:

- Flat
- Cylindrical
- Spherical
- Chrome

This section describes these wrapping types. In the examples, the direction vector (the v vector) lies along the z-axis, and the up vector (the u vector) lies along the y-axis, with the origin at [0 0 0].

Flat

The flat wrap conforms to the faces of an object as if the texture were a piece of rubber that was stretched over the object.

The [*u v*] coordinates are derived from a vector [*x y z*] by using the following equations:

u=sux–ou

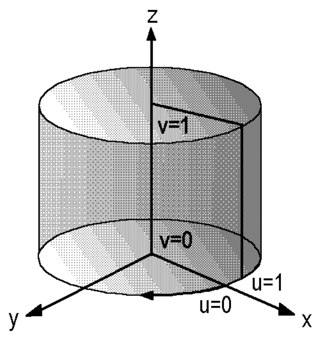
v=svy–ov

In these formulas, *s* is the window-scaling factor and *o* is the window origin. The application should choose a pair of scaling factors and offsets that map the ranges of *x* and *y* to the range from 0 to 1 for *u* and *v*.

Cylindrical

The cylindrical wrap treats the texture as if it were a piece of paper that is wrapped around a cylinder so that the left edge is joined to the right edge. The object is then placed in the middle of the cylinder and the texture is deformed inward onto the surface of the object.

For a cylindrical texture map, the effects of the various vectors are shown in the following illustration.



The direction vector specifies the axis of the cylinder, and the up vector specifies the point on the outside of the cylinder where *u* equals 0. To calculate the texture [*u v*] coordinates for a vector [*x y z*], the system uses the following equations:

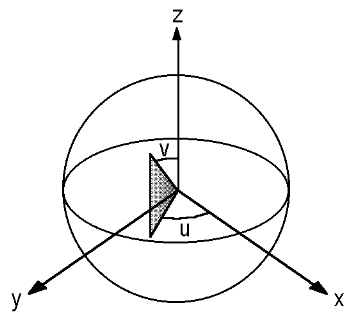
$$u = \frac{s_u}{2\pi}\tan^{-1}\frac{x}{y} - o_y$$

$$v = s_v z - o_v$$

Typically, *u* would be left unscaled and *v* would be scaled and translated so that the range of *z* maps to the range from 0 to 1 for *v*.

Spherical

For a spherical wrap, the u-coordinate is derived from the angle that the vector [*x y* 0] makes with the x-axis (as in the cylindrical map) and the v-coordinate from the angle that the vector [*x y z*] makes with the z-axis. Note that this mapping causes distortion of the texture at the z-axis.



This translates to the following equations:

$$u = \frac{Su}{2\pi} \tan^{-1} \frac{x}{y} - ou$$
$$v = \frac{Sv}{\pi} \tan^{-1} \frac{z}{\sqrt{x^2 + y^2 + z^2}} - ov$$

The scaling factors and texture origin will often not be needed here as the unscaled range of *u* and *v* is already 0 through 1.

Chrome

A chrome wrap allocates texture coordinates so that the texture appears to be reflected onto the objects. The chrome wrap takes the reference frame position and uses the vertex normals in the mesh to calculate reflected vectors. The texture u- and v-coordinates are then calculated from the intersection of these reflected vectors with an imaginary sphere that surrounds the mesh. This gives the effect of the mesh reflecting whatever is wrapped on the sphere.

Direct3D Retained-Mode Tutorial

To create a Windows-based Direct3D Retained-Mode application, you set up the features of two different environments: the devices, viewports, and color capabilities of the Windows environment, and the models, textures, lights, and positions of the virtual environment. This section describes some simple Retained-Mode sample code that is part of this SDK.

Setting up the Windows Environment

The Rmmain.cpp file in the samples provded with this SDK is used as a basis for all of the Retained-Mode samples. This file contains the standard Windows framework of initialization, setting up a message loop, and creating a window procedure for message processing, but it also does some work that is specific to Direct3D Retained-Mode applications:

- It enumerates the current Direct3D device drivers. This is described in the **Enumerating Direct3D Device Drivers** section.
- It creates the Retained-Mode API, the scene and camera frames, and a DirectDrawClipper object. This is described in the **Initialization** section.
- It creates a Direct3DRM device and viewport. This is described in the **Creating a Direct3DRM Device and Viewport** section.
- It sets the rendering quality, dithering flag, texture quality, shades, default texture colors, and default texture shades. This is described in the **Setting the Render State** section.
- It calls a locally defined BuildScene function to set up the lights and visuals in a scene. This is described in the **Setting Up the Virtual Environment** section.
- It renders the scene. This is described in the **Rendering into a Viewport** section.
- It makes the mouse status available to sample applications. This is described in the **OverrideDefaults and ReadMouse** section.

Enumerating Direct3D Device Drivers

Rmmain.cpp calls the locally defined InitApp function as its first action inside the WinMain function, and it fails if InitApp is not successful. InitApp creates the window and initializes all objects that are required to begin rendering a 3D scene.

After performing some standard tasks in the initialization of a Windows application and initializing global variables to acceptable default settings, InitApp verifies that Direct3D device drivers are present by calling the locally defined EnumDrivers function.

EnumDrivers sets up a callback function that examines each of the drivers in the system and picks the best one for the application's needs. This is not the easiest or even the best way to find Direct3D device drivers, however. Instead of setting up an enumeration routine, most applications will allow the system to automatically enumerate the current drivers by calling the **IDirect3DRM::CreateDeviceFromClipper** method and specifying NULL for the *lpGUID* parameter. This is the recommended way to create a Retained-Mode device because it always works, even if the user installs new hardware. When you use this method and both a hardware and a software device meet the default requirements, the system always retrieves the hardware device. An application should enumerate devices instead of specifying NULL for *lpGUID* only if it has unusual requirements.

EnumDrivers first calls the **DirectDrawCreate** function to create a DirectDraw object and then calls **IDirectDraw::QueryInterface** to retrieve an interface to Direct3D. The first parameter of this method is the interface identifier (IID) that identifies the **IDirect3D** COM interface. These identifiers are defined in the D3d.h header file. Each identifier is in the form IID_*InterfaceName*. For example, the IID for the **IDirect3DRMTexture** interface is IID_IDirect3DRMTexture.

After creating the Direct3D interface, EnumDrivers calls the **IDirect3D::EnumDevices** method to enumerate the Direct3D drivers. The first parameter to this method is a **D3DENUMDEVICESCALLBACK** callback function, in this case named enumDeviceFunc. The second parameter is a pointer to the myglobs structure, which contains the global variables Rmmain.cpp uses to set up the 3D environment. The **CurrDriver** member of the myglobs structure is simply an integer indicating the number of the Direct3D driver currently being used. The application initializes this value to -1 before calling **IDirect3D::EnumDevices**; the callback function checks this value to find out whether the driver being enumerated is the first valid driver.

```
static BOOL EnumDrivers(HWND win)
{
LPDIRECTDRAW lpDD;
LPDIRECT3D lpD3D;
HRESULT rval;
HMENU hmenu;

rval = DirectDrawCreate(NULL, &lpDD, NULL);
rval = lpDD->QueryInterface(IID_IDirect3D, (void**) &lpD3D);

myglobs.CurrDriver = -1;
rval = lpD3D->EnumDevices(enumDeviceFunc, &myglobs.CurrDriver);

lpD3D->Release();
lpDD->Release();

// Code omitted here that adds the driver names to the File menu.

return TRUE;
}
```

The enumDeviceFunc (**D3DENUMDEVICESCALLBACK**) callback function records each usable Direct3D device driver's name and globally unique identifier (GUID) and chooses a driver. The following code sample shows the enumDeviceFunc callback function. First, this callback function creates a **D3DDEVICEDESC** structure and copies the input *lpContext* parameter to a parameter called *lpStartDriver*. (The *lpContext* parameter is application-defined data—it can be any data the application requires.) The fourth parameter passed to enumDeviceFunc is a pointer to a **D3DDEVICEDESC** structure describing the hardware; the function uses the **dcmColorModel** member of this structure to determine whether it should examine the description of the hardware or the description of the hardware emulation layer (HEL). Any driver that cannot support the current bit-depth is skipped. The current driver's GUID and name are copied into the myglobs structure. Then the enumDeviceFunc callback function performs a few tests to compare the driver being enumerated against the driver stored as *lpStartDriver*. Whenever the driver being enumerated is implemented in hardware or uses the RGB instead of the monochromatic color model, that driver becomes the new *lpStartDriver*.

Finally, enumDeviceFunc increments the variable that counts the number of drivers. If this count exceeds the maximum, it returns D3DENUMRET_CANCEL to cancel the enumeration. Otherwise, it returns D3DENUMRET_OK to continue the enumeration.

```
static HRESULT WINAPI enumDeviceFunc(
    LPGUID lpGuid, LPSTR lpDeviceDescription, LPSTR lpDeviceName,
    LPD3DDEVICEDESC lpHWDesc, LPD3DDEVICEDESC lpHELDesc,
    LPVOID lpContext)
{
static BOOL hardware = FALSE; // Current start driver is hardware
static BOOL mono = FALSE;     // Current start driver is mono light
LPD3DDEVICEDESC lpDesc;
int *lpStartDriver = (int *)lpContext;

// Decide which device description we should consult.

lpDesc = lpHWDesc->dcmColorModel ? lpHWDesc : lpHELDesc;

// If this driver cannot render in the current display bit depth, skip
// it and continue with the enumeration.

if (!(lpDesc->dwDeviceRenderBitDepth & BPPToDDBD(myglobs.BPP)))
    return D3DENUMRET_OK;

// Record this driver's info.

memcpy(&myglobs.DriverGUID[myglobs.NumDrivers], lpGuid, sizeof(GUID));
strcpy(&myglobs.DriverName[myglobs.NumDrivers][0], lpDeviceName);

// Choose hardware over software, RGB lights over mono lights.

if (*lpStartDriver == -1) {

    // This is the first valid driver.

    *lpStartDriver = myglobs.NumDrivers;
    hardware = lpDesc == lpHWDesc ? TRUE : FALSE;
    mono = lpDesc->dcmColorModel & D3DCOLOR_MONO ? TRUE : FALSE;
} else if (lpDesc == lpHWDesc && !hardware) {

    // This driver is hardware and start driver is not.

    *lpStartDriver = myglobs.NumDrivers;
    hardware = lpDesc == lpHWDesc ? TRUE : FALSE;
    mono = lpDesc->dcmColorModel & D3DCOLOR_MONO ? TRUE : FALSE;
} else if ((lpDesc == lpHWDesc && hardware ) || (lpDesc == lpHELDesc
                        && !hardware)) {
    if (lpDesc->dcmColorModel == D3DCOLOR_MONO && !mono) {

    // This driver and start driver are the same type, and this
    // driver is mono while start driver is not.

    *lpStartDriver = myglobs.NumDrivers;
    hardware = lpDesc == lpHWDesc ? TRUE : FALSE;
    mono = lpDesc->dcmColorModel & D3DCOLOR_MONO ? TRUE : FALSE;
    }
}
myglobs.NumDrivers++;
if (myglobs.NumDrivers == MAX_DRIVERS)
    return (D3DENUMRET_CANCEL);
return (D3DENUMRET_OK);
}
```

When EnumDrivers function returns, it calls the **IDirect3D::Release** and **IDirectDraw::Release** methods to decrease the reference count of these COM objects. Every time an application calls the **Release** method on an object, the reference count for that object is reduced by 1. The system deallocates the object when its reference count reaches 0.

Whenever Rmmain.cpp calls a COM method, the return value is checked and the function fails if the method does not succeed. This error-checking has been omitted from the samples in this section for the sake of brevity. Sometimes the error-checking code calls the D3DRMErrorToString function (which is implemented in Rmerror.c in the same directory as Rmmain.cpp). This function simply maps an **HRESULT** error value to an explanatory string and returns the string.

If any COM interfaces have already been successfully created when a method fails, the error-checking code calls the **Release** method for that interface before returning FALSE.

Initialization

After enumerating the Direct3D device drivers, the InitApp function in Rmmain.cpp calls the **Direct3DRMCreate** function to create the Retained-Mode API:

```
LPDIRECT3DRM lpD3DRM;
HRESULT rval;

rval = Direct3DRMCreate(&lpD3DRM);
```

The sample then uses the Direct3DRM object to create the master reference frame (the "scene") and the camera frame by calling the **IDirect3DRM::CreateFrame** method. After creating the camera frame, this sample calls the **IDirect3DRMFrame::SetPosition** method to set its position.

```
rval = lpD3DRM->CreateFrame(NULL, &myglobs.scene);
rval = lpD3DRM->CreateFrame(myglobs.scene, &myglobs.camera);

// Set the position of the camera frame.

rval = myglobs.camera->SetPosition(myglobs.scene, D3DVAL(0.0),
    D3DVAL(0.0), D3DVAL(0.0));
```

The **scene** and **camera** members of the myglobs structure have the same type: **LPDIRECT3DRMFRAME**. The **D3DVAL** macro creates a value whose type is **D3DVALUE** from a supplied floating-point number.

Next, InitApp creates a DirectDraw clipper object by calling the **DirectDrawCreateClipper** function, and associates the window with the clipper object by calling the **IDirectDrawClipper::SetHWnd** method. This clipper object is subsequently used to create the Direct3D Windows device. The advantage of using a clipper object to create the device is that in this case DirectDraw does the clipping automatically.

```
LPDIRECTDRAWCLIPPER lpDDClipper;
HWND win;

rval = DirectDrawCreateClipper(0, &lpDDClipper, NULL);
rval = lpDDClipper->SetHWnd(0, win);
```

InitApp uses the DirectDraw clipper object and the Direct3D driver retrieved by the EnumDrivers function in a call to the locally defined CreateDevAndView function. CreateDevAndView is discussed in the following section, **Creating a Direct3DRM Device and Viewport**.

```
GetClientRect(win, &rc);
if (!CreateDevAndView(lpDDClipper, myglobs.CurrDriver,
        rc.right, rc.bottom)) {
    return FALSE;
}

// Create the scene to be rendered by calling this sample's BuildScene.

if (!BuildScene(myglobs.dev, myglobs.view, myglobs.scene,
    myglobs.camera))
    return FALSE;

// Ready to render.

myglobs.bInitialized = TRUE;

// Display the window.
```

```
ShowWindow(win, cmdshow);
UpdateWindow(win);

return TRUE;
```

Creating a Direct3DRM Device and Viewport

The InitApp function calls the locally defined CreateDevAndView function to create a Direct3DRM device and viewport. This function's parameters are a pointer to a DirectDrawClipper object, the number identifying the current driver, and the width and height of the current window's client rectangle.

After verifying that the window dimensions are valid, CreateDevAndView calls the **IDirect3DRM::CreateDeviceFromClipper** method to create a Direct3DRM device (stored in the **dev** member of the myglobs structure).

```
static BOOL CreateDevAndView(LPDIRECTDRAWCLIPPER lpDDClipper,
    int driver, int width, int height)
{
HRESULT rval;

if (!width || !height) {
    Msg("Cannot create a D3DRM device with invalid window dimensions.");
    return FALSE;
}

// Create the D3DRM device from this window and using the specified D3D
// driver.

rval = lpD3DRM->CreateDeviceFromClipper(lpDDClipper,
    &myglobs.DriverGUID[driver], width, height, &myglobs.dev);
```

CreateDevAndView uses the device it just created and the camera frame in a call to the **IDirect3DRM::CreateViewport** method that creates a Direct3DRM viewport. The width and height of the viewport are retrieved by calling the **IDirect3DRMDevice::GetWidth** and **IDirect3DRMDevice::GetHeight** methods. The viewport is stored in the **view** member of the myglobs structure. After creating the viewport, the code sets its background depth to a large number by calling the **IDirect3DRMViewport::SetBack** method.

```
width = myglobs.dev->GetWidth();
height = myglobs.dev->GetHeight();
rval = lpD3DRM->CreateViewport(myglobs.dev, myglobs.camera, 0, 0, width,
    height, &myglobs.view);

rval = myglobs.view->SetBack(D3DVAL(5000.0));
```

Finally, CreateDevAndView calls the locally defined SetRenderState function to set the rendering quality, fill mode, lighting state, and color shade. This function is described in **Setting the Render State**.

```
// Set the rendering quality, fill mode, lighting state and
// color shade information.

if (!SetRenderState())
    return FALSE;
return TRUE;
}
```

Setting the Render State

The InitApp function calls CreateDevAndView, which in turn calls the SetRenderState function to set the rendering quality, fill mode, lighting state, and color shade. Rmmain.cpp calls the SetRenderState function whenever the render state may have changed; for example, whenever the user changes the fill or lighting modes.

SetRenderState calls the **IDirect3DRMDevice::SetQuality** method to set the rendering quality, **IDirect3DRMDevice::SetDither** to set the dithering flag, and **IDirect3DRMDevice::SetTextureQuality** to set the texture quality. Each of these methods has a **Get** counterpart, which SetRenderState calls before setting any values. If the current value in the myglobs structure is the same as the value retrieved by the **Get** method, the function takes no further action.

```
BOOL SetRenderState(void)
{
HRESULT rval;

if (myglobs.dev->GetQuality() != myglobs.RenderQuality) {
    rval = myglobs.dev->SetQuality(myglobs.RenderQuality);
}

if (myglobs.dev->GetDither() != myglobs.bDithering) {
    rval = myglobs.dev->SetDither(myglobs.bDithering);
}

if (myglobs.dev->GetTextureQuality() != myglobs.TextureQuality) {
    rval = myglobs.dev->SetTextureQuality(myglobs.TextureQuality);
}
```

Now SetRenderState sets the shades, default texture colors, and default texture shades based on the current number of bits per pixel.

- If the device supports only one bit per pixel, SetRenderState uses **IDirect3DRMDevice::SetShades** to set the number of shades to four; this is the number of shades in a ramp of colors used for shading. SetRenderState also calls

IDirect3DRM::SetDefaultTextureShades to set the default number of shades for textures.

- If the device supports 16 bits per pixel, SetRenderState calls an additional method: **IDirect3DRM::SetDefaultTextureColors**. This method sets the default number of colors used in textures.
- If the device supports 24 or 32 bits per pixel, SetRenderState calls the same methods as for 16 bits per pixel. The only difference is that in this case it sets the shades and default texture shades to 256 instead of 32.

```
switch (myglobs.BPP) {
    case 1:
    if (FAILED(myglobs.dev->SetShades(4)))
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureShades(4)))
        goto shades_error;
    break;
    case 16:
    if (FAILED(myglobs.dev->SetShades(32)))
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureColors(64)))
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureShades(32)))
        goto shades_error;
    break;
    case 24:
    case 32:
    if (FAILED(myglobs.dev->SetShades(256)))
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureColors(64)))
        goto shades_error;
    if (FAILED(lpD3DRM->SetDefaultTextureShades(256)))
        goto shades_error;
    break;
}
return TRUE;

shades_error:
Msg("A failure occurred while setting color shade information.\n");
return FALSE;
}
```

The FAILED macro is defined as follows in the Winerror.h header file:

```
#define FAILED(Status) ((HRESULT)(Status)<0)
```

Setting Up the Virtual Environment

After the InitApp function has created the window, enumerated the Direct3D drivers, created the Retained-Mode API, created the scene and camera frames, created a DirectDrawClipper object, and created the Direct3D device and viewport, it calls a locally defined BuildScene function to build the scene. Each of the Direct3D samples in this SDK includes its own BuildScene function. These implementations of BuildScene vary depending on the capabilities being demonstrated by the sample.

This section discusses the simplest implementation of BuildScene in this SDK: the version in Egg.c. In this module, BuildScene performs the following tasks:

- Sets up the lights. This is described in the **Setting Up the Lights** section.
- Adds a mesh to the scene and sets up the position, rotation, and orientation of its frame. This is described in the **Loading and Adding a Mesh** section.
- Releases the objects that it has created. This is described in the **Releasing the Direct3DRM Objects** section.
- Overrides several of the default settings from Rmmain.cpp. This is described in the **OverrideDefaults and ReadMouse** section.

Setting Up the Lights

The BuildScene function implements two light sources: an ambient light and a directional light. An ambient light casts a uniform light over the entire scene. The position and orientation of its containing frame are ignored. A directional light, on the other hand, has direction but no position. It is attached to a frame but appears to illuminate all objects with equal intensity, as if it were at an infinite distance from each object.

First the BuildScene function creates a frame for the directional light and calls the **IDirect3DRMFrame::SetPosition** method to set its position in the scene. Then it calls the **IDirect3DRM::CreateLightRGB** method to create a directional light source. The **IDirect3DRMFrame::AddLight** method adds the directional light to the lighting frame. Then the **IDirect3DRMFrame::SetRotation** method causes the lighting frame to rotate.

After setting up the directional light, BuildScene creates a much dimmer ambient light source, again by calling **IDirect3DRM::CreateLightRGB**. Because ambient lights are not affected by the position and orientation of their containing frame, the BuildScene function specifies the frame for the entire scene in the call to **IDirect3DRMFrame::AddLight** that adds the ambient light.

```
LPDIRECT3DRMFRAME lights = NULL;
LPDIRECT3DRMLIGHT light1 = NULL;
LPDIRECT3DRMLIGHT light2 = NULL;
    .
    .
    .
if (FAILED(lpD3DRM->lpVtbl->CreateFrame(lpD3DRM, scene, &lights)))
    goto generic_error;
```

```
if (FAILED(lights->lpVtbl->SetPosition(lights, scene, D3DVAL(5),
        D3DVAL(5), -D3DVAL(1))))
    goto generic_error;
if (FAILED(lpD3DRM->lpVtbl->CreateLightRGB(lpD3DRM,
        D3DRMLIGHT_DIRECTIONAL, D3DVAL(0.9),
        D3DVAL(0.8), D3DVAL(0.7), &light1)))
    goto generic_error;
if (FAILED(lights->lpVtbl->AddLight(lights, light1)))
    goto generic_error;
if (FAILED(lights->lpVtbl->SetRotation(lights, scene, D3DVAL(0),
        D3DVAL(1), D3DVAL(1), D3DVAL(-0.02))))
    goto generic_error;
if (FAILED(lpD3DRM->lpVtbl->CreateLightRGB(lpD3DRM, D3DRMLIGHT_AMBIENT,
        D3DVAL(0.1), D3DVAL(0.1), D3DVAL(0.1), &light2)))
    goto generic_error;
if (FAILED(scene->lpVtbl->AddLight(scene, light2)))
    goto generic_error;
```

The generic_error label identifies a place in the code where Egg.c displays a message box with an error value, releases any created COM objects, and returns FALSE from the BuildScene function.

Loading and Adding a Mesh

The BuildScene function in Egg.c loads a simple egg-shaped mesh into the scene and rotates the frame containing the mesh.

A call to the **IDirect3DRM::CreateMeshBuilder** method retrieves the **IDirect3DRMMeshBuilder** COM interface. BuildScene uses this interface for only one call before releasing it; it calls the **IDirect3DRMMeshBuilder::Load** method to load a mesh file named Egg.x.

Next, BuildScene calls **IDirect3DRM::CreateFrame** to create a frame for the mesh, and then it calls **IDirect3DRMFrame::AddVisual** to add the mesh to that frame.

Then, BuildScene sets the position and orientation of the camera frame in relation to the scene frame by calling the **IDirect3DRMFrame::SetPosition** and **IDirect3DRMFrame::SetOrientation** methods. Finally, it rotates the mesh frame in relation to the scene frame by calling the **IDirect3DRMFrame::SetRotation** method.

```
LPDIRECT3DRMMESHBUILDER egg_builder = NULL;
LPDIRECT3DRMFRAME egg = NULL;
    .
    .
    .
if (FAILED(lpD3DRM->lpVtbl->CreateMeshBuilder(lpD3DRM, &egg_builder)))
    goto generic_error;
rval = egg_builder->lpVtbl->Load(egg_builder, "egg.x", NULL,
    D3DRMLOAD_FROMFILE, NULL, NULL);

if (FAILED(lpD3DRM->lpVtbl->CreateFrame(lpD3DRM, scene, &egg)))
    goto generic_error;

if (FAILED(egg->lpVtbl->AddVisual(
        egg, (LPDIRECT3DRMVISUAL)egg_builder)))
    goto generic_error;

if (FAILED(camera->lpVtbl->SetPosition(
        camera, scene, D3DVAL(0), D3DVAL(0), -D3DVAL(10))))
    goto generic_error;
if (FAILED(camera->lpVtbl->SetOrientation(
        camera, scene, D3DVAL(0), D3DVAL(0), D3DVAL(1), D3DVAL(0),
        D3DVAL(1), D3DVAL(0))))
    goto generic_error;
if (FAILED(egg->lpVtbl->SetRotation(
        egg, scene, D3DVAL(0), D3DVAL(1), D3DVAL(1), D3DVAL(0.02))))
    goto generic_error;
```

Releasing the Direct3DRM Objects

The last task performed by the BuildScene function in Egg.c is to release the Direct3DRM objects it creates—the frames for the mesh and lights, the mesh itself, and the two lights:

```
RELEASE(egg);
RELEASE(lights);
RELEASE(egg_builder);
RELEASE(light1);
RELEASE(light2);
```

The RELEASE macro calls the **Release** method for the appropriate interface. It is defined in C++ and C versions in the Rmdemo.h header file. The syntax of the C version looks like this:

```
#define RELEASE(x) if (x != NULL) {x->lpVtbl->Release(x); x = NULL;}
```

Rmmain.cpp also releases any objects it creates, in its CleanUpAndPostQuit function. The code calls CleanUpAndPostQuit whenever initialization has been completed and the application must exit: when a WM_QUIT message is received, when the user chooses the Exit command, on fatal errors, and so on.

```
void CleanUpAndPostQuit(void)
{
myglobs.bInitialized = FALSE;
RELEASE(myglobs.scene);
RELEASE(myglobs.camera);
RELEASE(myglobs.view);
RELEASE(myglobs.dev);
RELEASE(lpD3DRM);
RELEASE(lpDDClipper);
myglobs.bQuit = TRUE;
}
```

OverrideDefaults and ReadMouse

Samples can use the OverrideDefaults function with Rmmain.cpp, in addition to the standard BuildScene function. Egg.c includes an implementation of OverrideDefaults that simply sets the **bNoTextures** member to TRUE in the Defaults structure (defined in Rmdemo.h) and changes the name of the application:

```
void OverrideDefaults(Defaults *defaults)
{
    defaults->bNoTextures = TRUE;
    strcpy(defaults->Name, "Egg Direct3DRM Example");
}
```

Rmmain.cpp includes another function, ReadMouse, that samples can use to retrieve the mouse status. For an example of how this function is used, see the Quat.c sample in this SDK.

```
void ReadMouse(int* b, int* x, int* y)
{
    *b = myglobs.mouse_buttons;
    *x = myglobs.mouse_x;
    *y = myglobs.mouse_y;
}
```

Rendering into a Viewport

The Rmmain.cpp file calls the RenderLoop function as part of the message loop in the WinMain function. The RenderLoop function performs a few simple tasks:

- Calls **IDirect3DRMFrame::Move** to apply the rotations and velocities for all frames in the hierarchy.
- Calls **IDirect3DRMViewport::Clear** to clear the current viewport, setting it to the current background color.
- Calls **IDirect3DRMViewport::Render** to render the current scene into the current viewport.
- Calls **IDirect3DRMDevice::Update** to copy the rendered image to the display.

```
static BOOL RenderLoop()
{
HRESULT rval;
rval = myglobs.scene->Move(D3DVAL(1.0));
rval = myglobs.view->Clear();
rval = myglobs.view->Render(myglobs.scene);
rval = myglobs.dev->Update();
return TRUE;
}
```

Direct3D Immediate-Mode Overview

About Immediate Mode

This section describes Direct3D's Immediate Mode, Microsoft's low-level 3D API. Direct3D's 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's Retained Mode is built on top of Immediate Mode.

Developers who use Immediate Mode instead of Retained Mode are typically experienced in high-performance programming issues, and may also be experienced in 3D graphics. Even if this is the case for you, you should read **A Technical Foundation for 3D Programming**. This section discusses implementation details of Direct3D that you need to know to work effectively with the system. The overall Direct3D architecture is described in **Direct3D Architecture**; this is essential reading for Immediate-Mode developers. If you want an overview of Immediate Mode, you should read **Introduction to Direct3D Immediate-Mode Objects**. Your best source of information about Immediate Mode, however, is probably the sample code included with this SDK; it illustrates how to put Direct3D's Immediate Mode to work in real-world applications.

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

Introduction to Direct3D Immediate-Mode Objects

Direct3D's Immediate Mode consists of API elements that create objects, fill them with data, and link them together. Direct3D's Retained Mode is built on top of Immediate Mode. For a description of the overall organization of the system and the organization of Immediate Mode in particular, see **Direct3D Architecture**.

The following table shows the eight object types in Immediate Mode, their Component Object Model (COM) interfaces, and a description of each:

Object type	**COM interface and description**
Interface	**IDirect3D**

	COM interface object
Device	**IDirect3DDevice**
	Hardware device
Texture	**IDirect3DTexture**
	DirectDraw surface containing an image
Material	**IDirect3DMaterial**
	Surface properties, such as color and texture
Light	**IDirect3DLight**
	Light source
Viewport	**IDirect3DViewport**
	Screen region to draw to
Matrix	**IDirect3DDevice**
	4-by-4 homogeneous transformation matrix
ExecuteBuffer	**IDirect3DExecuteBuffer**
	List of vertex data and instructions about how to render it

Rendering is done by using execute buffers. These buffers contain vertex data and a list of opcodes that, when interpreted, instruct the rendering engine to produce an image. The execute buffer COM object contains only pointers and some descriptions of the format of the buffer. The contents of the buffer are dynamically allocated and can reside in the memory of the graphics card.

Each of the objects can effectively exist in one or more of the following forms:

- A COM object.
- A structure exposed to the developer that effectively contains a copy of the data in the COM object. This form is typically used to copy data into or out of the actual COM object.
- A handle. In this case, the data resides on the hardware and can be manipulated by it.

The following table shows the forms in which each of the Direct3D objects can exist:

	COM interface	**Structure**	**Handle**
Device	x		
Texture	x	x	x
Material	x	x	x
Light	x	x	
Viewport	x		
Matrix		x	x

ExecuteBuffer	x	x

Direct3D Object Types

This section describes Direct3D object types. An application creates Direct3D objects roughly in the following order:

Direct3D objects

Device objects

Texture objects

Material objects

Light objects

Viewport objects

Execute-buffer objects

Direct3D Interface Objects

You can create a Direct3D interface object by calling the **IDirectDraw::QueryInterface** method, as follows:

```
lpDirectDraw->QueryInterface(
    IID_IDirect3D,  // IDirect3D interface ID
    lpD3D);         // Address of a Direct3D object
```

The object referred to by an **IDirect3D** interface contains a list of viewports, lights, materials, and devices. You can use the methods of the **IDirect3D** interface to create other objects or to find Direct3D devices.

Direct3D Device Objects

You can create a Direct3D device object by calling the **IDirectDrawSurface::QueryInterface** method for a backbuffer surface. The following example calls the **IDirectDraw::CreateSurface** and **IDirectDrawSurface::GetAttachedSurface** methods to retrieve the backbuffer surface.

```
lpDirectDraw->CreateSurface(
    lpDDSurfDesc,   // Address of a DDSURFACEDESC structure
    lpFrontBuffer,  // Address of a DIRECTDRAWSURFACE structure
    pUnkOuter);     // NULL
lpFrontBuffer->GetAttachedSurface(
    &ddscaps,         // Address of a DDSCAPS structure
    &lpBackBuffer);   // Address of a DIRECTDRAWSURFACE structure
lpBackBuffer->QueryInterface(
    GUIDforID3DDevice,   // ID for IDirect3DDevice interface
    lpD3DDevice);        // Address of a DIRECT3DDEVICE object
```

The first parameter of the call to the **IDirectDrawSurface::QueryInterface** method for the back buffer is the globally unique identifier (GUID) for the **IDirect3DDevice** interface. You can retrieve this GUID by calling the **IDirect3D::EnumDevices** method; the system supplies the GUID when it calls the **D3DENUMDEVICESCALLBACK** callback function you supply in the call to **IDirect3D::EnumDevices**.

A Direct3D device object resides on (or "is owned by") the interface list and has its own list of execute buffers and viewports. It also has a list of textures and materials, each of which points both to the next texture or material in the list and back to the device. For more information about this hierarchy, see **Object Connectivity**.

The methods of the **IDirect3DDevice** interface report hardware capabilities, maintain a list of viewports, manipulate matrix objects, and execute execute-buffer objects.

Matrices appear to you only as handles. 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. Matrix handles are used in execute buffers.

Direct3D Texture Objects

A texture is a rectangular array of colored pixels. (The rectangle does not necessarily have to be square, although 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-, 24-, and 32-bit textures. If you use the monochromatic (or ramp) color model, however, you can use only 8-bit textures.

You can retrieve an interface to a Direct3D texture object by calling the **IDirectDrawSurface::QueryInterface** method and specifying IID_IDirect3DTexture. An **IDirect3DTexture** interface is actually an interface to a DirectDrawSurface object, not to a distinct Direct3D texture object. For more information about the relationship between textures in Direct3D and surfaces in DirectDraw, see **Direct3D Texture Interface**.

The following example demonstrates how to create an **IDirect3DTexture** interface and then how to load the texture by calling the **IDirect3DTexture::GetHandle** and **IDirect3DTexture::Load** methods.

```
lpDDS->QueryInterface(IID_IDirect3DTexture,
    lpD3DTexture);  // Address of a DIRECT3DTEXTURE object
lpD3DTexture->GetHandle(
    lpD3DDevice,    // Address of a DIRECT3DDEVICE object
    lphTexture);    // Address of a D3DTEXTUREHANDLE
lpD3DTexture->Load(
    lpD3DTexture);  // Address of a DIRECT3DTEXTURE object
```

Texture objects reside on the interface list and point both to the next texture in a device's list and back to their associated device or devices. (For more information about this hierarchy, see **Object Connectivity**.) The texture handle is used in materials and execute buffers, and as a z-buffer for a viewport. You can use the **IDirect3DTexture** interface to load and unload textures, retrieve handles, and track changes to palettes.

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. For more information, see **Direct3DRMWrap**.

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.

For more information about wrapping, see **Wrapping Types** in the introduction to Direct3D Retained-Mode objects.

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_DSTBLEND** render states to specify how colors in the source and destination are combined. The combination options (called *blend factors*) 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. Mipmapping is a computationally low-cost way of improving the quality of rendered textures. Each prefiltered image, or level, in the mipmap is a power of two smaller than the previous level.

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.

For more information about how to use DirectDraw to create mipmaps, see **Mipmaps**.

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 DirectDraw's 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's support for color keys, see **Color Keying**.

Direct3D Material Objects

You can create an interface to a Direct3D material by calling the **IDirect3D::CreateMaterial** method. The following example demonstrates how to create an **IDirect3DMaterial** interface. Then it demonstrates how to set the material and retrieve its handle by calling the **IDirect3DMaterial::SetMaterial** and **IDirect3DMaterial::GetHandle** methods.

```
lpDirect3D->CreateMaterial(
    lplpDirect3DMaterial,  // Address of a new material
    pUnkOuter);            // NULL
lpDirect3DMaterial->SetMaterial(
    lpD3DMat);             // Address of a D3DMATERIAL structure
lpDirect3DMaterial->GetHandle(
    lpD3DDevice,           // Address of a DIRECT3DDEVICE object
    lpD3DMat);             // Address of a D3DMATERIAL structure
```

Material objects reside on the interface list and point both to the next material in a device's list and back to their associated device or devices. (For more information about this hierarchy, see **Object Connectivity**.) A material contains colors and may contain a texture handle. The material handle is used inside execute buffers or to set the background of a viewport. You can use the **IDirect3DMaterial** interface to get and set materials, retrieve handles, and reserve colors.

Direct3D Light Objects

You can create a Direct3D light object by calling the **IDirect3D::CreateLight** method. The following example demonstrates how to create an **IDirect3DLight** interface, and then it sets the light by calling the **IDirect3DLight::SetLight** method.

```
lpDirect3D->CreateLight(
    lplpDirect3DLight,  // Address of a new light
    pUnkOuter);         // NULL
lpDirect3DLight->SetLight(
    lpLight);    // Address of a D3DLIGHT structure
```

Light objects reside on the interface list and on a viewport list. You can use the **IDirect3DLight** interface to get and set lights.

Direct3D Viewport Objects

You can create a Direct3D viewport object by calling the **IDirect3D::CreateViewport** method. The following example demonstrates how to create an **IDirect3DViewport** interface. Then it demonstrates how to add the viewport to a device by calling the **IDirect3DDevice::AddViewport** method and how to set up the viewport by calling the **IDirect3DViewport::SetViewport**, **IDirect3DViewport::SetBackground**, and **IDirect3DViewport::AddLight** methods.

```
lpDirect3D->CreateViewport(
    lplpDirect3DViewport,  // Address of a new viewport
    pUnkOuter);            // NULL
lpD3DDevice->AddViewport(
    lpD3DViewport)         // Attach viewport to device
lpD3DViewport->SetViewport(
    lpData);               // Address of a D3DVIEWPORT structure that
                           // Sets the viewport's location on the screen
lpD3DViewport->SetBackground(
    lphMat);               // Address of a D3DMATERIALHANDLE for
                           // background
lpD3DViewport->AddLight(
    lpD3DLight);           // Address of a light object
```

Viewport objects reside on the interface and device lists. The object maintains a list of lights as well as screen data, and it may have a material handle and a texture handle for the background. You can use the **IDirect3DViewport** interface to get and set backgrounds and viewports, add and delete lights, and transform vertices.

Direct3D Execute-Buffer Objects

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. For a description of execute buffers, see **Execute Buffers**.

You can create a Direct3D execute-buffer object 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-buffer objects 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:

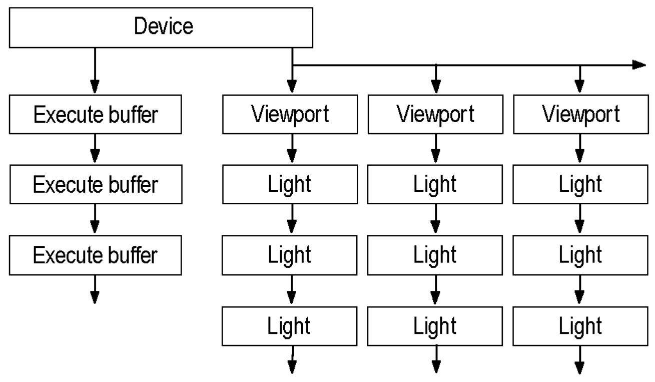
```
lpD3DExBuf->Lock(
    lpDesc);.       // Address of a DIRECT3DEXECUTEBUFFERDESC structure
//   .
//   .  Store contents through the supplied address
//   .
lpD3DExBuf->Unlock();
lpD3DExBuf->SetExecuteData(
    lpData);        // Address of a D3DEXECUTEDATA structure
```

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.

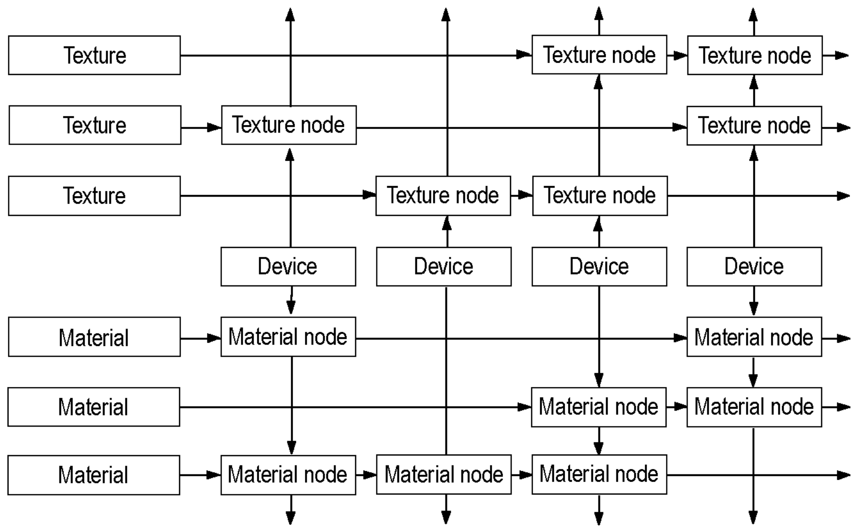
You can use the **IDirect3DExecuteBuffer** interface to get and set execute data, and to lock, unlock, optimize, and validate the execute buffer.

Object Connectivity

The following illustration shows how each execute buffer and viewport is owned by exactly one device, and each light is owned by exactly one viewport.



Material and texture objects can be associated with more than one device. Each node in the following illustration contains a pointer back to the head of the list it is in. (These pointers, however, are not shown in the illustration.) From any texture or material node, pointers lead back both to the texture or material object and to the relevant device.



You can create interface, device, and texture objects by calling the **QueryInterface** method. Material, light, and viewport objects are created by calls to the methods of the **IDirect3D** interface. Execute buffers and matrices are created by calls to the methods of the **IDirect3DDevice** interface. An interface object (Direct3D object) has a list of all created device, viewport, light, and material objects, but not execute buffers or textures.

Direct3D Immediate-Mode Tutorial

To create a Windows-based Direct3D Immediate-Mode application, you create DirectDraw and Direct3D objects, set render states, fill execute buffers, and execute those buffers. This section explains some simple Immediate-Mode sample code that is part of this SDK.

The D3dmain.cpp file in the samples provided with this SDK is used as a basis for all of the other Immediate-Mode samples. D3dmain.cpp contains the standard Windows framework of initialization, setting up a message loop, and creating a window procedure for message processing, but it also does some work that is specific to Direct3D Immediate-Mode applications. This work is discussed in the following sections:

- **Beginning Initialization**
- **Creating DirectDraw and Direct3D Objects**
- **Setting Up the Device-Creation Callback Function**
- **Initializing the Viewport**
- **Setting the Immediate-Mode Render State**
- **Completing Initialization**

- **Running the Rendering Loop**
- **Cleaning Up**

The Immediate-Mode samples in this SDK include some code that is not documented here. In particular, this SDK includes a collection of helper functions collectively referred to as the D3DApp functions. You might find them useful when writing your own Immediate-Mode applications. These helper functions are referred to frequently in this documentation but are not covered exhaustively. They are implemented by the D3dapp.c, Ddcalls.c, D3dcalls.c, Texture.c, and Misc.c source files. The Stats.cpp source file sends frame-rate and screen-mode information to the screen.

Every sample that uses D3dmain.cpp must implement the following functions, which enable samples to customize their behavior:

- InitScene
- InitView
- RenderScene
- ReleaseView
- ReleaseScene
- OverrideDefaults

In addition, samples can use the SetMouseCallback and SetKeyboardCallback functions to read mouse and keyboard input.

Beginning Initialization

The first Direct3D task the WinMain function in D3dmain.cpp does is call the locally defined AppInit function, which creates the application window and initializes all objects needed to begin rendering. The WinMain function also implements the message pump for D3dmain.cpp and calls the locally defined RenderLoop and CleanUpAndPostQuit functions. The AppInit function calls other functions to help it do its work, and these functions, in turn, call still more functions; this group of initialization functions is the subject of most of this tutorial.

After performing some standard tasks in the initialization of a Windows application and initializing global variables to acceptable default settings, AppInit calls the InitScene function that each sample that uses D3dmain.cpp must implement. Simple sample applications, such as Oct1.c, implement versions of InitScene that do nothing but return TRUE. More complicated samples, such as Tunnel.c, use InitScene to allocate memory, generate points, and set global variables.

The last thing the AppInit function does before it returns is call the CreateD3DApp function, which is implemented in D3dmain.cpp. The functions called by CreateD3Dapp do much of the initialization work.

Creating DirectDraw and Direct3D Objects

The CreateD3DApp function in D3dmain.cpp is responsible for initializing the DirectDraw and Direct3D objects that are required before rendering begins. Important local functions called by CreateD3DApp include D3DAppCreateFromHWND, D3DAppGetRenderState, OverrideDefaults, D3DAppSetRenderState, ReleaseView, and InitView. Functions whose names begin with D3DApp are part of the D3DApp series of helper functions.

The same command-line options passed to the WinMain function are also passed to CreateD3DApp. Valid options are **-systemmemory** and **-emulation**. The **-systemmemory** option is used purely for debugging. The **-emulation** option prevents the application from using DirectDraw or Direct3D hardware.

CreateD3DApp calls the D3DAppAddTexture function to generate a series of textures. Then the D3DAppAddTexture function creates a source texture surface and object in system memory. Then it creates a second, initially empty, texture surface in video memory if hardware is present. The source texture is loaded into the destination texture surface and then discarded. This two-stage process allows a device to compress or reformat a texture map as it enters video memory. The code uses the **IDirectDrawSurface::QueryInterface** method to retrieve an interface to an **IDirect3DTexture** interface and the **IDirect3DTexture::Load** method to load the textures. The **IDirect3DTexture::GetHandle** method is used to create a list of texture handles.

After creating a list of textures, CreateD3DApp creates the DirectDraw and Direct3D objects required to start rendering. The code uses the D3DAppCreateFromHWND helper function. D3DAppCreateFromHWND uses functions that are implemented in the D3dapp.c, D3dcalls.c, Texture.c, and Ddcalls.c source files.

First, D3DAppCreateFromHWND uses the **DirectDrawEnumerate** and **DirectDrawCreate** functions to create and initialize the DirectDraw object, sets the values of global variables, and enumerates the display modes by calling the **IDirectDraw2::EnumDisplayModes** method.

D3DAppCreateFromHWND then creates the Direct3D object and enumerates the Direct3D device drivers. To create the Direct3D object, it calls the **IDirectDraw::QueryInterface** method, passing the IID_IDirect3D interface identifier. It uses **IDirect3D::EnumDevices** to enumerate the device drivers.

Calling **IDirect3D::EnumDevices** is not the easiest or even the best way to find Direct3D device drivers, however. Instead of setting up an enumeration routine, most Immediate-Mode applications use the **IDirect3D::FindDevice** method. This method allows you to simply specify the capabilities of the device you prefer—the system examines the available drivers and returns the globally unique identifier (GUID) for the first matching device. The system always searches the hardware first, so if both a hardware and a software device meet the required capabilities, the system returns the GUID for the hardware device.

After choosing a device driver and display mode (full-screen versus windowed), D3DAppCreateFromHWND creates front and back buffers for the chosen display mode. The code performs different actions based on whether the application is running in a window or the full screen, and whether video memory or system memory is being employed. If the application is running in a window, the code calls the **IDirectDraw::CreateClipper** method to create a clipper object and then calls the **IDirectDrawClipper::SetHWnd** method to attach it to the window and the **IDirectDrawSurface::SetClipper** method to attach it to the front buffer.

Then, D3DAppCreateFromHWND checks whether the front buffer is palettized. If it is, the code initializes the palette. First it creates the palette by calling the **IDirectDraw::CreatePalette** method, then it uses the **IDirectDrawSurface::SetPalette** method to set this as the palette for the front and back surfaces.

At this point the code calls the **IDirectDraw::CreateSurface** method to create a z-buffer, the **IDirectDrawSurface::AddAttachedSurface** method to attach the z-buffer to the back buffer, and the **IDirectDrawSurface::GetSurfaceDesc** method to determine whether the z-buffer is in video memory.

Then the code creates an **IDirect3DDevice** interface and uses it to enumerate the texture formats. The sample calls the **IDirectDrawSurface::QueryInterface** method to create the interface and the **IDirect3DDevice::EnumTextureFormats** method to enumerate the texture formats. When the textures have been enumerated, the code uses the same series of calls it employed in CreateD3DApp to load the textures and create a list of texture handles.

After using the driver's bit-depth and total video memory to filter the appropriate display modes, the code sets up the device-creation callback function, which is described in **Setting Up the Device-Creation Callback Function**.

When the device-creation callback function has been set up, D3DAppCreateFromHWND sets the application's render state, which is described in **Setting the Immediate-Mode Render State**.

When the D3DAppCreateFromHWND function has created the required Direct3D objects and set up the render state, it is almost finished. The function calls a local function to set the dirty rectangles for the front and back buffers to the entire client area, sets flags indicating that the application is initialized and that rendering can proceed, and returns TRUE.

The last part of the D3DAppCreateFromHWND function is its error-handling section. Whenever a call fails, the error-handling code jumps to the exit_with_error label. This section calls the callback function that destroys the device, resets the display mode and cooperative level if the application was running in full-screen mode, releases all the Direct3D and DirectDraw objects that were created, and returns FALSE.

Setting Up the Device-Creation Callback Function

The third parameter of the D3DAppCreateFromHWND function is an address of a callback function that is implemented as the AfterDeviceCreated function in D3dmain.cpp. AfterDeviceCreated creates the Direct3D viewport and returns it to D3DAppCreateFromHWND.

First the code calls the **IDirect3D::CreateViewport** method to create a viewport, and then it calls the **IDirect3DDevice::AddViewport** method to add the viewport to the newly created Direct3D device. After initializing the viewport's dimensions in a **D3DVIEWPORT** structure, the code calls the **IDirect3DViewport::SetViewport** method to set the viewport to those dimensions.

Then, the AfterDeviceCreated function calls the InitView function. InitView, like the InitScene function called earlier by D3dmain.cpp, must be implemented by each sample that uses D3dmain.cpp. One implementation of InitView is described in **Initializing the Viewport**.

After calling InitView and changing some of the menu items, AfterDeviceCreated finishes by calling the CleanUpAndPostQuit function. This function is described in **Cleaning Up**.

Initializing the Viewport

Each of the code samples that uses D3dmain.cpp must implement a version of the InitView function to set up the viewport and create the sample's execute buffers. This section discusses the implementation of InitView found in the Oct1.c sample.

First, InitView creates and initializes some materials, material handles, and texture handles. It uses the **IDirect3D::CreateMaterial** method to create a material, the **IDirect3DMaterial::SetMaterial** method to set the material data that InitView has just initialized, and the **IDirect3DMaterial::GetHandle** and **IDirect3DViewport::SetBackground** methods to set this material as the background for the viewport.

Now the InitView function sets the view, world, and projection matrices for the viewport. InitView uses the MAKE_MATRIX macro to create and set matrices. MAKE_MATRIX is defined in D3dmacs.h as follows:

```
#define MAKE_MATRIX(lpDev, handle, data) \
    if (lpDev->lpVtbl->CreateMatrix(lpDev, &handle) != D3D_OK) \
        return FALSE; \
    if (lpDev->lpVtbl->SetMatrix(lpDev, handle, &data) != D3D_OK) \
        return FALSE
```

As you can see, MAKE_MATRIX is simply a convenient way of calling the **IDirect3DDevice::CreateMatrix** and **IDirect3DDevice::SetMatrix** methods in a single step.

Now InitView creates and sets up an execute buffer. After initializing the members of a **D3DEXECUTEBUFFERDESC** structure, the code calls the **IDirect3DDevice::CreateExecuteBuffer** method to create the execute buffer and **IDirect3DExecuteBuffer::Lock** to lock it so that it can be filled.

InitView fills the execute buffer by using the OP_STATE_TRANSFORM and STATE_DATA macros from D3dmacs.h. These macros are described in **Setting the Immediate-Mode Render State**, along with more information about working with execute buffers.

When the execute buffer has been set up, InitView calls the **IDirect3DExecuteBuffer::Unlock** method to unlock it, the **IDirect3DExecuteBuffer::SetExecuteData** method to set the data into the buffer, and then the **IDirect3DDevice::BeginScene**, **IDirect3DDevice::Execute**, and **IDirect3DDevice::EndScene** methods to execute the execute buffer. Because the function has no further use for this execute buffer, it then calls **IDirect3DExecuteBuffer::Release**.

The InitView function now sets up two more materials by using the same procedure it used to set the materials earlier: it uses the **IDirect3D::CreateMaterial** method to create a material, the **IDirect3DMaterial::SetMaterial** method to set the material data (after filling the members of the **D3DMATERIAL** structure), and the **IDirect3DMaterial::GetHandle** method to retrieve a handle to the material. These handles are used later with the **D3DLIGHTSTATE_MATERIAL** member of the **D3DLIGHTSTATETYPE** enumerated type to associate lights with the new materials.

Now InitView sets up the vertices. The code uses the **D3DVALP** macro to convert floating-point numbers into the **D3DVALUE** members of the **D3DVERTEX** structure. It also uses the **D3DRMVectorNormalize** function to normalize the x-coordinate of the normal vector for each of the vertices it sets up.

When the vertices have been set up, InitView creates another execute buffer, copies the vertices to the buffer, and sets the execute data. It does not execute the execute buffer, however; this time, executing the execute buffer occurs in the rendering loop.

Finally, InitView sets up the lights for Oct1.c. After initializing the members of a **D3DLIGHT** structure, it calls the **IDirect3D::CreateLight**, **IDirect3DLight::SetLight**, and **IDirect3DViewport::AddLight** methods to add the light to the viewport.

Setting the Immediate-Mode Render State

The D3DAppISetRenderState function in the D3dcalls.c source file creates and executes an execute buffer that sets the render state and light state for the current viewport. The D3DAppCreateFromHWND function calls D3DAppISetRenderState from D3dapp.c; generally, the sample code calls D3DAppISetRenderState whenever the render state needs to be set or reset. This section reproduces the D3DAppISetRenderState function (except for some error-checking code).

After setting some local variables, including the **D3DEXECUTEBUFFERDESC** and **D3DEXECUTEDATA** structures, D3DAppISetRenderState calls the **IDirect3DDevice::CreateExecuteBuffer** method to create an execute buffer. When the execute buffer has been created, the code calls the **IDirect3DExecuteBuffer::Lock** method to lock it so that it can be filled.

```
BOOL D3DAppISetRenderState()
{
D3DEXECUTEBUFFERDESC debDesc;
D3DEXECUTEDATA d3dExData;
LPDIRECT3DEXECUTEBUFFER lpD3DExCmdBuf = NULL;
LPVOID lpBuffer, lpInsStart;
size_t size;

// Create an execute buffer of the required size and lock it
// so that it can be filled.

size = 0;
size += sizeof(D3DINSTRUCTION) * 3;
size += sizeof(D3DSTATE) * 17;
memset(&debDesc, 0, sizeof(D3DEXECUTEBUFFERDESC));
debDesc.dwSize = sizeof(D3DEXECUTEBUFFERDESC);
debDesc.dwFlags = D3DDEB_BUFSIZE;
debDesc.dwBufferSize = size;

LastError = d3dappi.lpD3DDevice->lpVtbl->CreateExecuteBuffer(
    d3dappi.lpD3DDevice, &debDesc, &lpD3DExCmdBuf, NULL);

LastError = lpD3DExCmdBuf->lpVtbl->Lock(lpD3DExCmdBuf, &debDesc);
memset(debDesc.lpData, 0, size);

lpInsStart = debDesc.lpData;
lpBuffer = lpInsStart;
```

The *d3dappi.lpD3DDevice* parameter in the call to **IDirect3DDevice::CreateExecuteBuffer** is a pointer to a Direct3DDevice object. The **lpData** member of the *debDesc* variable (a **D3DEXECUTEBUFFERDESC** structure) is a pointer to the actual data in the execute buffer.

Now the D3DAppISetRenderState function sets the render states. It uses the OP_STATE_DATA macro to help do some of this work. This macro, in turn, uses the PUTD3DINSTRUCTION macro. These macros are defined as follows in the D3dmacs.h header file in this SDK:

```
#define PUTD3DINSTRUCTION(op, sz, cnt, ptr) \
    ((LPD3DINSTRUCTION) ptr)->bOpcode = op; \
    ((LPD3DINSTRUCTION) ptr)->bSize = sz; \
    ((LPD3DINSTRUCTION) ptr)->wCount = cnt; \
    ptr = (void *)(((LPD3DINSTRUCTION) ptr) + 1)
#define OP_STATE_RENDER(cnt, ptr) \
    PUTD3DINSTRUCTION(D3DOP_STATERENDER, sizeof(D3DSTATE), cnt, ptr)
```

Notice that the PUTD3DINSTRUCTION macro is little more than an initializer for the **D3DINSTRUCTION** structure. D3DOP_STATERENDER, the first parameter of the call to PUTD3DINSTRUCTION in the OP_STATE_RENDER macro, is one of the opcodes in the **D3DOPCODE** enumerated type; the second parameter is the size of the **D3DSTATE** structure.

The STATE_DATA macro, which is also defined in D3dmacs.h, handles the render states. To work with the render states, it uses pointers to members in the **D3DSTATE** structure—in particular, a pointer to the **D3DRENDERSTATETYPE** enumerated type.

```
#define STATE_DATA(type, arg, ptr) \
    ((LPD3DSTATE) ptr)->drstRenderStateType = (D3DRENDERSTATETYPE)type; \
    ((LPD3DSTATE) ptr)->dwArg[0] = arg; \
    ptr = (void *)(((LPD3DSTATE) ptr) + 1)
```

The following code fragment from D3DAppISetRenderState uses the OP_STATE_RENDER and STATE_DATA macros to set 14 render states. The d3dapprs structure is the D3DAppRenderState structure, which is defined in the D3dapp.h header file.

```
OP_STATE_RENDER(14, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_SHADEMODE, d3dapprs.ShadeMode, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_TEXTUREPERSPECTIVE,
        d3dapprs.bPerspCorrect, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ZENABLE, d3dapprs.bZBufferOn &&
        d3dappi.ThisDriver.bDoesZBuffer, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ZWRITEENABLE, d3dapprs.bZBufferOn,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ZFUNC, D3DCMP_LESSEQUAL, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_TEXTUREMAG, d3dapprs.TextureFilter,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_TEXTUREMIN, d3dapprs.TextureFilter,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_TEXTUREMAPBLEND, d3dapprs.TextureBlend,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_FILLMODE, d3dapprs.FillMode, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_DITHERENABLE, d3dapprs.bDithering,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_SPECULARENABLE, d3dapprs.bSpecular,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ANTIALIAS, d3dapprs.bAntialiasing,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_FOGENABLE, d3dapprs.bFogEnabled,
        lpBuffer);
    STATE_DATA(D3DRENDERSTATE_FOGCOLOR, d3dapprs.FogColor, lpBuffer);
```

Now the OP_STATE_RENDER and STATE_DATA macros set three light states. The OP_EXIT macro simply uses the PUTD3DINSTRUCTION macro to call the **D3DOP_EXIT** opcode from the **D3DOPCODE** enumerated type.

```
OP_STATE_LIGHT(3, lpBuffer);
    STATE_DATA(D3DLIGHTSTATE_FOGMODE, d3dapprs.bFogEnabled ?
        d3dapprs.FogMode : D3DFOG_NONE, lpBuffer);
    STATE_DATA(D3DLIGHTSTATE_FOGSTART,
        *(unsigned long*)&d3dapprs.FogStart, lpBuffer);
    STATE_DATA(D3DLIGHTSTATE_FOGEND, *(unsigned long*)&d3dapprs.FogEnd,
        lpBuffer);
OP_EXIT(lpBuffer);
```

Now that the render states have been set, D3DAppISetRenderState unlocks the execute buffer by calling the **IDirect3DExecuteBuffer::Unlock** method. It sets the execute data by calling the **IDirect3DExecuteBuffer::SetExecuteData** method. Finally, it begins the scene, executes the execute buffer, and ends the scene again by calling the **IDirect3DDevice::BeginScene**, **IDirect3DDevice::Execute**, and **IDirect3DDevice::EndScene** methods.

```
LastError = lpD3DExCmdBuf->lpVtbl->Unlock(lpD3DExCmdBuf);

memset(&d3dExData, 0, sizeof(D3DEXECUTEDATA));
d3dExData.dwSize = sizeof(D3DEXECUTEDATA);
d3dExData.dwInstructionOffset = (ULONG) 0;
d3dExData.dwInstructionLength = (ULONG) ((char*)lpBuffer -
    (char*)lpInsStart);
lpD3DExCmdBuf->lpVtbl->SetExecuteData(lpD3DExCmdBuf, &d3dExData);
LastError =
    d3dappi.lpD3DDevice->lpVtbl->BeginScene(d3dappi.lpD3DDevice);
LastError =
    d3dappi.lpD3DDevice->lpVtbl->Execute(d3dappi.lpD3DDevice,
        lpD3DExCmdBuf, d3dappi.lpD3DViewport);
LastError = d3dappi.lpD3DDevice->lpVtbl->EndScene(d3dappi.lpD3DDevice);
```

The D3DAppISetRenderState has finished its work with the execute buffer, so it calls the **IDirect3DExecuteBuffer::Release** method to release it, and it returns.

```
lpD3DExCmdBuf->lpVtbl->Release(lpD3DExCmdBuf);
return TRUE;
}
```

Completing Initialization

The CreateD3DApp function that is called by the AppInit function in WinMain has created most of the underpinnings of a Direct3D application, but it hasn't finished yet. Before D3dmain.cpp calls the rendering loop, CreateD3DApp must complete a few more tasks.

After copying the current render states into the application's local D3DAppRenderState structure, CreateD3DApp calls the OverrideDefaults function. OverrideDefaults is another function that must be supported by all samples that use D3dmain.cpp. Sometimes an application will do very little in its OverrideDefaults function; for example, the Oct1.c sample simply replaces the default name with the string, "Octagon D3D Example".

Now CreateD3DApp calls the D3DAppSetRenderState function, which simply checks the status of the saved render states and either resets them (if no render states were provided in the call) or saves them before calling the D3DAppISetRenderState function, which is discussed in **Setting the Immediate-Mode Render State**.

As a final step before entering the rendering loop, CreateD3DApp calls the ReleaseView and InitView functions. These functions, like OverrideDefaults, are implemented by each sample application. ReleaseView simply releases any objects created in a previous call to the InitView function. (InitView, as you recall, is first called as part of the AfterDeviceCreated callback function.) This final call to InitView sets up the viewport again (in case there have been changes since the device was created) and re-creates the sample's execute buffers. For more information about InitView, see **Initializing the Viewport**.

This is the end not only of the CreateD3DApp function in D3dmain.cpp; it is also the end of the AppInit function. Now that the code has finished with the initialization part of the work, it can enter the rendering loop.

Running the Rendering Loop

After initializing the application, the WinMain function in D3dmain.cpp sets up the message loop. The code monitors the message queue until there are no pressing messages and then calls the RenderLoop function. RenderLoop is defined in D3dmain.cpp. WinMain maintains a count of the number of times RenderLoop can fail; if it fails more than three times, the application quits.

The RenderLoop function renders the next frame in the scene and updates the window.

The first task in the RenderLoop function is to restore any lost DirectDraw surfaces, if possible. It calls the **IDirectDrawSurface::IsLost** method to identify lost surfaces and **IDirectDrawSurface::Restore** to restore them. Then RenderLoop clears the back buffer and, if enabled, the z-buffer, by calling the **IDirect3DRMViewport::Clear** method.

Now RenderLoop calls the RenderScene function implemented by the sample application calling D3dmain.cpp. A sample's RenderScene function can take into account a complex series of special cases, or it can be as simple as the implementation in Oct1.c. Then, the RenderScene function executes the execute buffer by calling **IDirect3DDevice::BeginScene**, **IDirect3DDevice::Execute**, and **IDirect3DDevice::EndScene**, retrieves any new execute data by calling **IDirect3DExecuteBuffer::GetExecuteData**, updates the screen extents, and then calls a local TickScene function that changes the viewing position by calling the **IDirect3DDevice::SetMatrix** method.

Finally, the RenderLoop function calls the D3DAppRenderExtents helper function to keep track of the changed sections of the front and back buffers and blits or flips the back buffer to the front buffer.

Cleaning Up

Whenever an error occurs from which the application cannot recover, or the application receives a WM_QUIT or MENU_EXIT message, the application calls the CleanUpAndPostQuit function. CleanUpAndPostQuit does some simple error-checking and then calls the ReleaseScene function. ReleaseScene is the last of the functions that must be implemented by sample applications using D3dmain.cpp. This is an application's chance to release any remaining objects or free memory. For simple applications, like Oct1.c, ReleaseScene is simply a stub.

Finally, the CleanUpAndPostQuit function calls the **PostQuitMessage** function to end the application.