Direct3D Immediate Mode

[This is preliminary documentation and subject to change.]

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

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

About Direct3D Immediate Mode

[This is preliminary documentation and subject to change.]

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

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

Microsoft developed the Direct3D Immediate Mode as a low-level 3-D API. Immediate Mode is ideal for developers who need to port games and other high-performance multimedia applications to the Microsoft Windows operating system. Immediate Mode is a device-independent way for applications to communicate with accelerator hardware at a low level. Direct3D Retained Mode is built on top of Immediate Mode.

These are some of the advanced features of Direct3D:

- Switchable depth buffering (using z-buffers or w-buffers)
- Flat and Gouraud shading
- Multiple lights and light types
- Full material and texture support, including mipmapping

- Robust software emulation drivers
- Transformation and clipping
- Hardware independence
- Full support on Windows NT/Windows 2000
- Support for the Intel MMX architecture

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

Why Use Direct3D Immediate Mode?

[This is preliminary documentation and subject to change.]

The world management of Immediate Mode is based on vertices, polygons, and commands that control them. It allows immediate access to the transformation, lighting, and rasterization 3-D graphics pipeline. If hardware isn't present to accelerate rendering, Direct3D offers applications a choice of software emulation drivers. Developers with existing 3-D applications and developers who need to achieve maximum performance by maintaining the thinnest possible layer between their application and the hardware should use Immediate Mode instead of Retained Mode.

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

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

Getting Started with Immediate Mode

[This is preliminary documentation and subject to change.]

The following sections describe some of the technical concepts you need to understand before you write programs that incorporate 3-D graphics. This is not a discussion of broad architectural details, nor is it an in-depth analysis of specific Direct3D components. For information about these topics, see Direct3D Immediate Mode Architecture and Direct3D Immediate Mode Essentials.

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

Information in this section is divided into the following groups:

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

3-D Coordinate Systems and Geometry

[This is preliminary documentation and subject to change.]

Programming Direct3D applications requires a working familiarity of 3-D geometric principles. This section introduces the most important geometric concepts needed for creating 3-D scenes in the following sections:

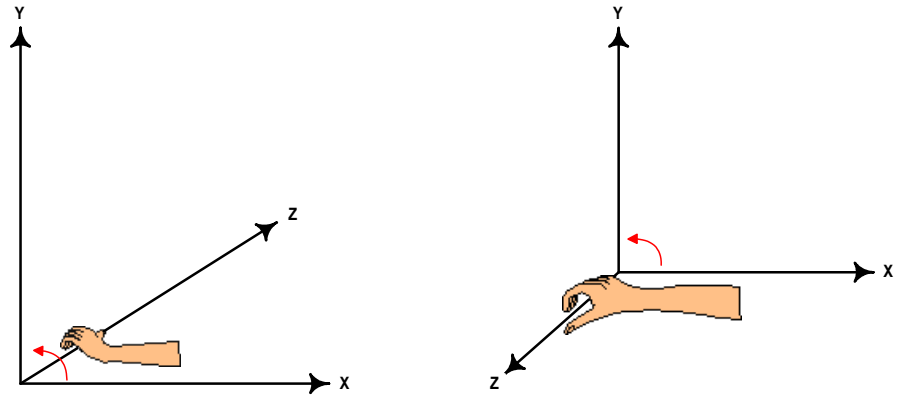
- 3-D Coordinate Systems
- 3-D Primitives
- Triangle Rasterization Rules
- Shading

The discussions in these topics are intended to provide you with a simple, high-level, understanding of the basic concepts employed by a Direct3D application. For a list of sources for much more detailed information about these topics, see Further Reading.

3-D Coordinate Systems

[This is preliminary documentation and subject to change.]

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



Direct3D uses a left-handed coordinate system. If you are porting an application that 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.
- Use the view matrix to scale world space by -1 in the z-direction. To do this, flip the sign of the _31, _32, _33, and _34 member of the **D3DMATRIX** structure that you use for your view matrix. (Likewise, Visual Basic applications can perform this operation on the corresponding members of the **D3DMATRIX** type.)

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

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

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

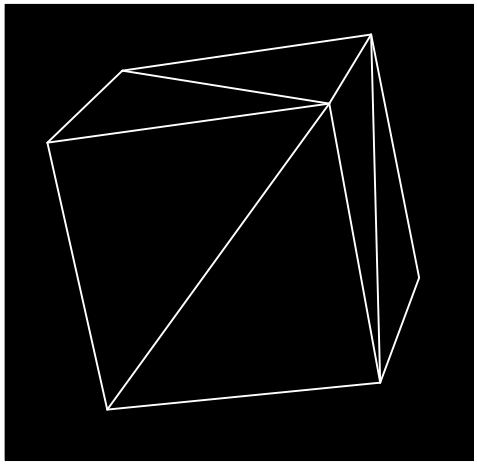
3-D Primitives

[This is preliminary documentation and subject to change.]

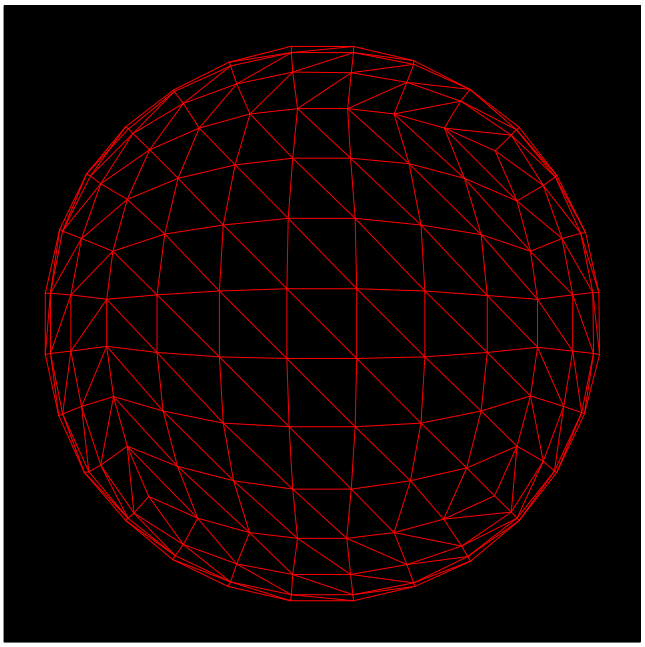
A 3-D primitive is a collection of vertices that form a single 3-D entity. The simplest primitive is a collection of points in a 3-D coordinate system, which is called a point list in Direct3D.

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

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



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



Triangle Rasterization Rules

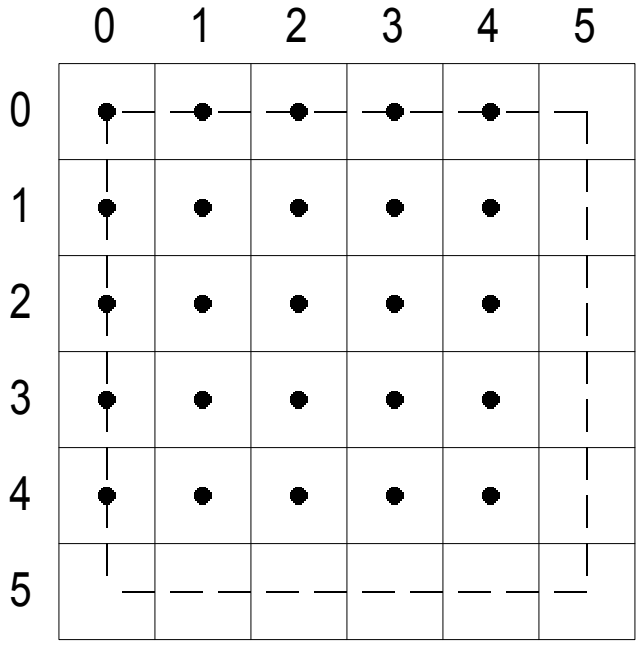
[This is preliminary documentation and subject to change.]

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

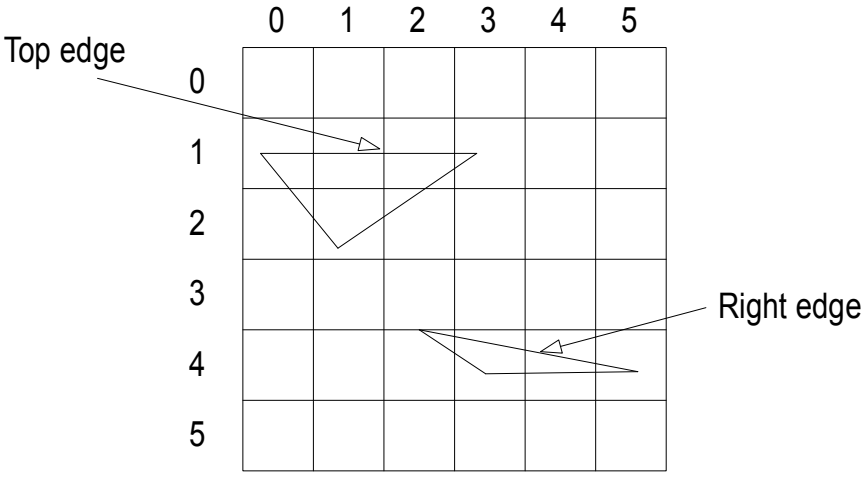
Direct3D uses a top-left filling convention for filling geometry. This is the same convention that is used for rectangles in GDI and OpenGL. In Direct3D the center of the pixel is the point at which decisions are made. If the center is inside a triangle, the pixel is part of the triangle. Pixel centers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

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

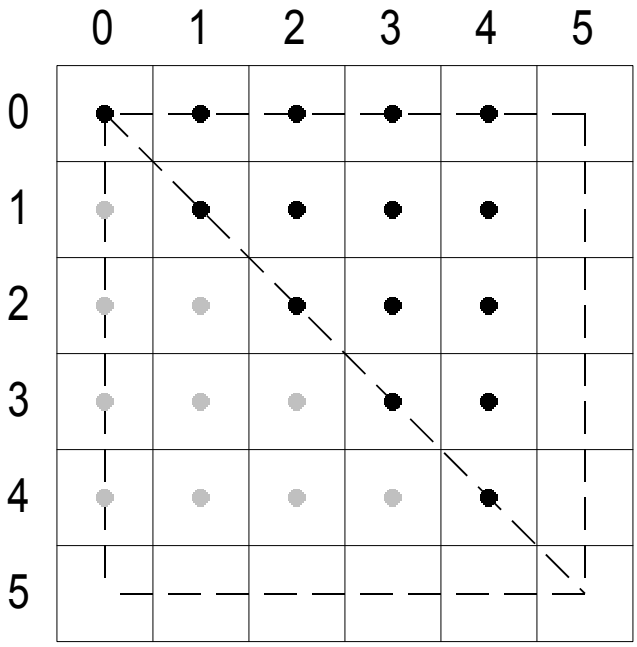
The following illustration shows a rectangle whose upper-left corner is at (0, 0) and whose lower-right corner is at (5, 5). This rectangle fills 25 pixels, just as you would expect. The width of the rectangle is defined as right minus left. The height is defined as bottom minus top.



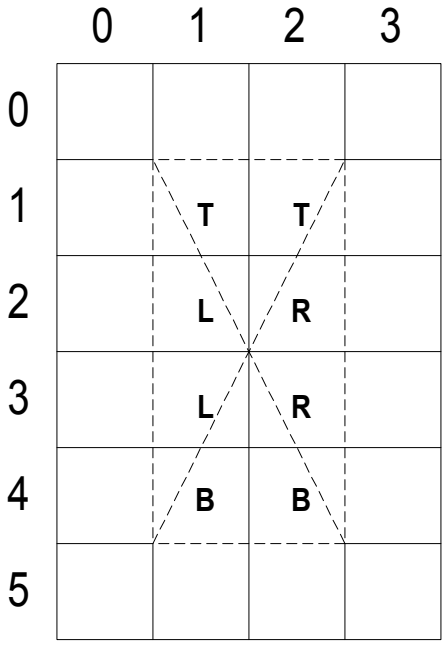
In the top-left filling convention, the word "top" refers to the vertical location of horizontal spans, and the word "left" refers to the horizontal location of pixels within a span. An edge cannot be a top edge unless it is horizontal—in the general case, most triangles will have only left and right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



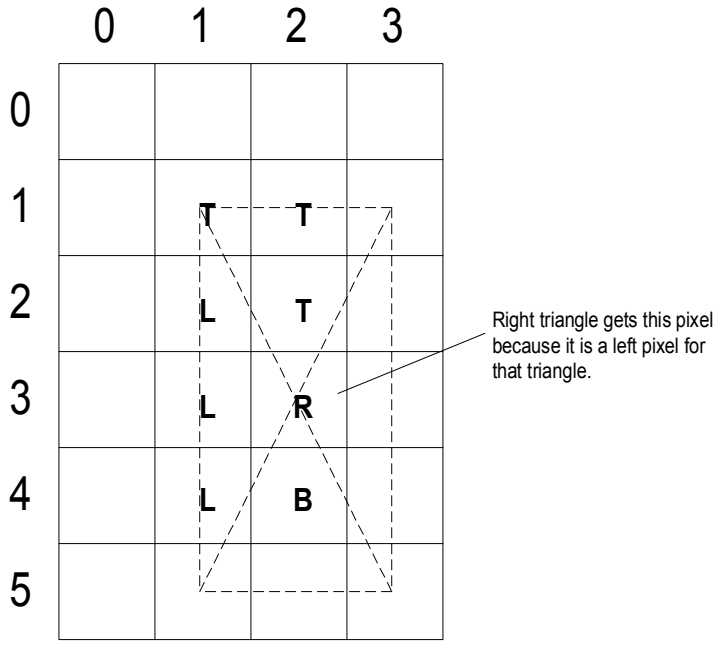
The top-left filling convention determines the action taken by Direct3D when a triangle passes through the center of a pixel. The following illustration shows two triangles, one at (0, 0), (5, 0), and (5, 5), and the other at (0, 5), (0, 0), and (5, 5). The first triangle in this case gets 15 pixels,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



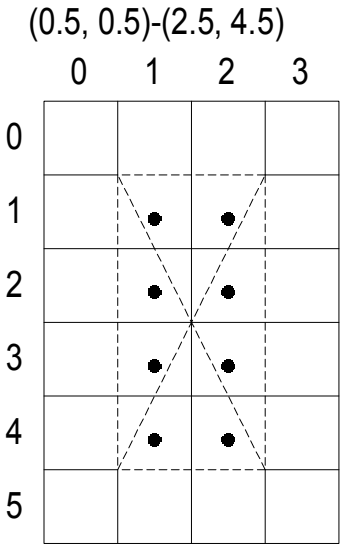
If, for example, you define a rectangle with its upper-left corner at (0.5, 0.5) and its lower-right corner at (2.5, 4.5), the center point of this rectangle would be at (1.5, 2.5). When the Direct3D rasterizer tessellates this rectangle, the center of each pixel would be unambiguously inside each of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

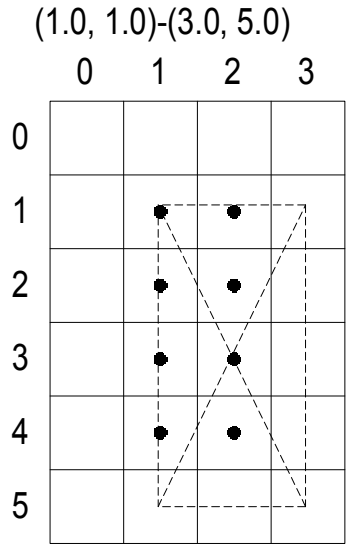


If you move the rectangle in the previous example so that its upper-left corner is at (1.0, 1.0), its lower-right corner at (3.0, 5.0), and its center point at (2.0, 3.0), Direct3D applies the top-left filling convention. Most of the pixels in this rectangle would straddle the border between two or more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



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





Shading

[This is preliminary documentation and subject to change.]

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

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

Shade Modes

[This is preliminary documentation and subject to change.]

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

- Flat Shading
- Gouraud Shading

Flat Shading

[This is preliminary documentation and subject to change.]

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

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

Gouraud Shading

[This is preliminary documentation and subject to change.]

When Direct3D renders a polygon using Gouraud shading, it computes a color for each vertex by using the vertex normal and lighting parameters. Then, it interpolates the color across the face of the polygons (See Face and Vertex Normal Vectors). The interpolation is done linearly. For example, if the red component of the color of vertex 1 is 0.8 and the red component of vertex 2 is 0.4, utilizing the Gouraud shade mode and the RGB color model, the Direct3D lighting module would assign a red component of 0.6 to the pixel at the midpoint of the line between these vertices.

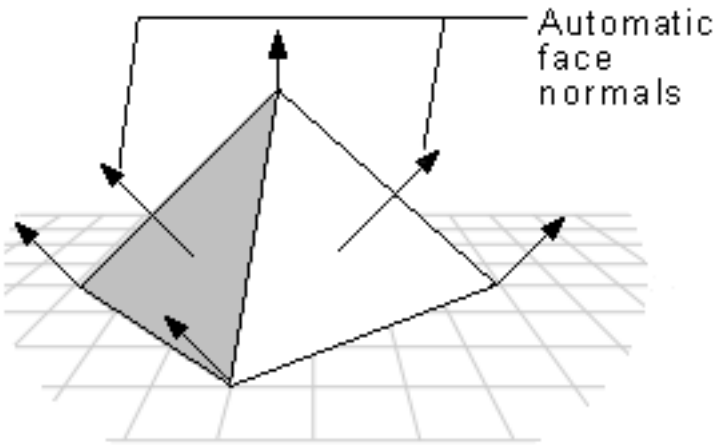
The following figure demonstrates Gouraud shading. This teapot is composed of many flat, triangular polygons. However, Gouraud shading makes the surface of the object appear curved and smooth.

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

Comparing Shading Modes

[This is preliminary documentation and subject to change.]

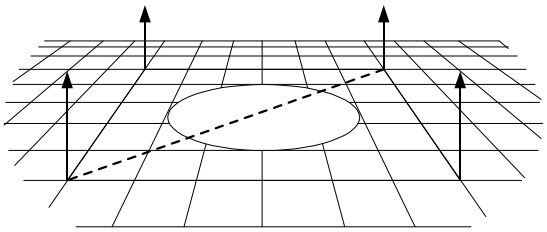
In the flat shade mode, the following pyramid would be displayed with a sharp edge between adjoining faces. In the Gouraud shade mode, however, shading values would be interpolated across the edge, and the final appearance would be of a curved surface.



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

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



In this case, 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.

Setting the Shade Mode

[This is preliminary documentation and subject to change.]

[C++]

Direct3D allows one shade mode to be selected at a time. By default, Gouraud shading is selected. In C++, the shade mode can be changed by calling the **IDirect3DDevice3::SetRenderState** method. The *dwRenderStateType* parameter should be set to D3DRENDERSTATE_SHADEMODE. The *dwRenderState* parameter must be set to a member of the **D3DSHADEMODE** enumeration. The following sample code fragments illustrate how the current shade mode of a Direct3D application can be set to flat or Gouraud shading mode.

```
// Set to flat shading.
// This code fragment assumes that lpDev3 is a valid pointer to
// an IDirect3DDevice3 interface.
hr = lpDev3->SetRenderState(D3DRENDERSTATE_SHADEMODE, D3DSHADE_FLAT);
if(FAILED(hr))
{
   // Code to handle the error goes here.
}

// Set to Gouraud shading (this is the default for Direct3D).
hr = lpDev3->SetRenderState(D3DRENDERSTATE_SHADEMODE,
                  D3DSHADE_GOURAUD);
if(FAILED(hr))
{
   // Code to handle the error goes here.
}
```

[Visual Basic]

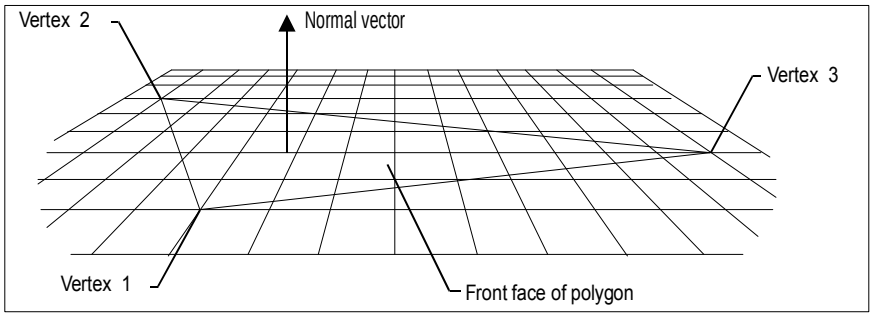
Direct3D allows one shade mode to be selected at a time. By default, Gouraud shading is selected. In Visual Basic, the shade mode can be changed by calling the **Direct3DDevice3.SetRenderState** method. The *state* parameter should be set to D3DRENDERSTATE_SHADEMODE. The *renderstate* parameter must be set to a member of the **CONST_D3DSHADEMODE** enumeration. The following code fragment illustrates how the current shade mode of a Direct3D application can be set to flat or Gouraud shading mode.

```
' Set to flat shading.
' This code fragment assumes that Dev3 is a valid reference to
' a Direct3DDevice3 object.
On Local Error Resume Next
Call Dev3.SetRenderState(D3DRENDERSTATE_SHADEMODE, _
                         D3DSHADE_FLAT)

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

' Set to Gouraud shading (this is the default for Direct3D).
Call Dev3.SetRenderState(D3DRENDERSTATE_SHADEMODE, _
                         D3DSHADE_GOURAUD)

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

Face and Vertex Normal Vectors

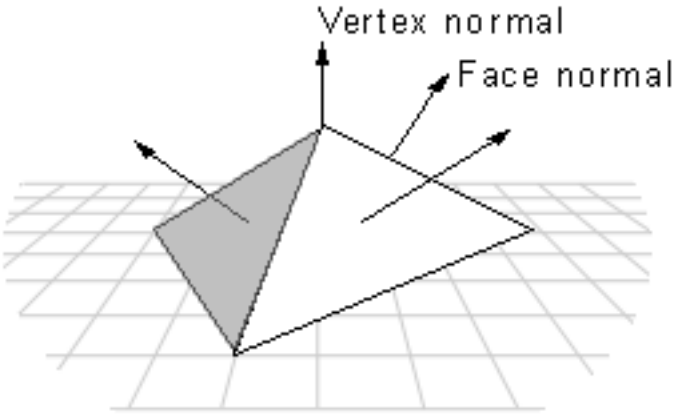
[This is preliminary documentation and subject to change.]

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



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

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



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

```
D3DVERTEX lpVertices[3];

// A vertex can be specified one structure member at a time.
lpVertices[0].x = 0;
lpVertices[0].y = 5;
```

```
lpVertices[0].z = 5;
lpVertices[0].nx = 0;    // X component of the normal vector.
lpVertices[0].ny = 0;    // Y component of the normal vector.
lpVertices[0].nz = -1;   // Points the normal back at the origin.
lpVertices[0].tu = 0;    // Only used if a texture is being used.
lpVertices[0].tv = 0;    // Only used if a texture is being used.

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

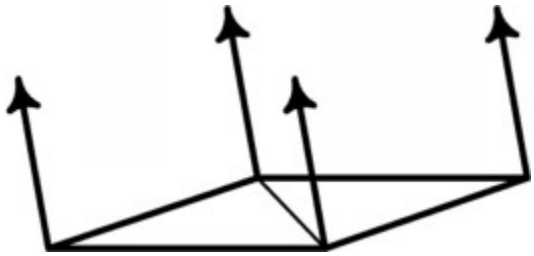
[Visual Basic]

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

```
D3DVERTEX Vertices(3)

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

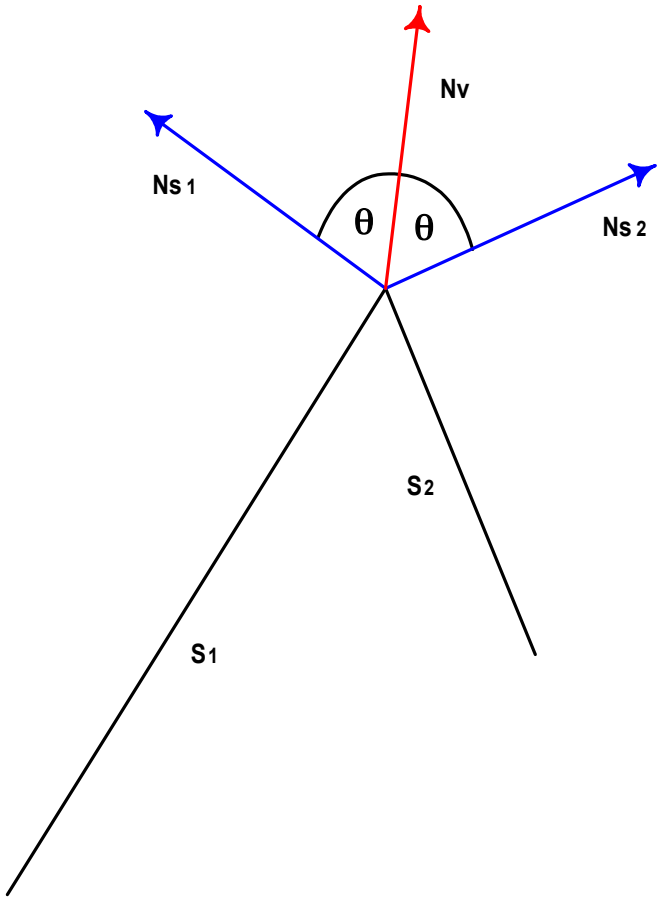
When applying Gouraud shading to a polygon, Direct3D uses the vertex normals to calculate the angle between the light source and the surface. It calculates the color and intensity values for the vertices and interpolates them for every point across all of the primitive's surfaces. Direct3D calculates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

If you are creating an object that is flat, then the vertex normals should be set to point perpendicular to the surface. The following illustration demonstrates this. A flat surface composed of two triangles is defined. The vertex normals all point perpendicular to the surface.

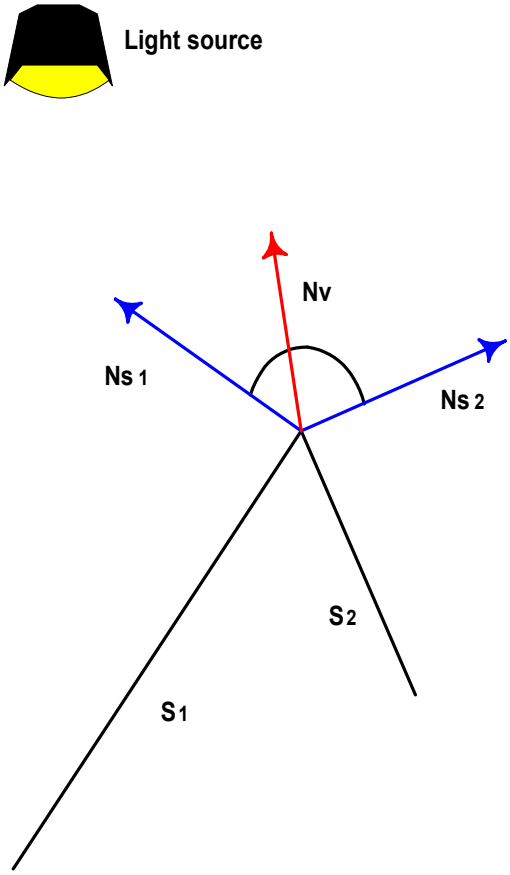


It is more likely, however, that your object is made up of triangle strips, and that the triangles are not coplanar. One simple way to get smooth shading across all of the triangles in the strip is to first calculate the surface normal vector for each polygonal face with which the vertex is associated.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

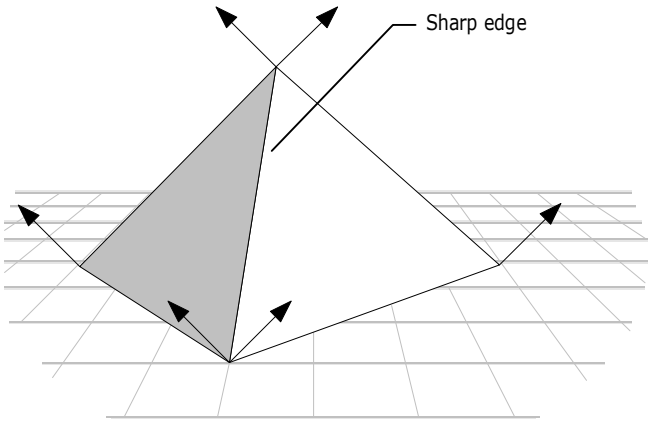
This method is illustrated by the following figure, which shows two surfaces, S1 and S2 seen edge-on from above. The normal vectors for S1 and S2 are shown in blue. The vertex normal vector is shown in red. The angle that the vertex normal vector makes with the surface normal of S1 is the same as the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



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



You can use the Gouraud shade mode to display some of the objects in a 3-D scene with sharp edges. If you are using execute buffers, your application needs to duplicate the vertex normal vectors at any intersection of faces where a sharp edge was required, as shown in the following illustration.



[C++]
If you are using the **IDirect3DDevice3::DrawPrimitive** or **IDirect3DDevice3::DrawIndexedPrimitive** methods to render your scene, you must define the object with sharp edges as a triangle list, rather than a triangle strip. When you define an object as a triangle strip, Direct3D treats it as a single polygon composed of multiple triangular faces. Gouraud shading is applied both across each face of the polygon and between adjacent faces. The result is an object that is smoothly shaded from face to face. Since a triangle list is a polygon composed of a series of disjoint triangular faces, Direct3D applies Gouraud shading across each face of the polygon. However it is not applied from face to face. If two or more triangles of a triangle list are adjacent, they appear to have a sharp edge between them.

[Visual Basic]
If you are using the **Direct3DDevice3.DrawPrimitive** or **Direct3DDevice3.DrawIndexedPrimitive** methods to render your scene, you must define the object with sharp edges as a triangle list, rather than a triangle strip. When you define an object as a triangle strip, Direct3D treats it as a single polygon composed of multiple triangular faces. Gouraud shading is applied both across each face of the polygon and between adjacent faces. The result is an object that is smoothly shaded from face to face. Since a triangle list is a polygon composed of a series of disjoint triangular faces, Direct3D applies Gouraud shading across each face of the polygon. However it is not applied from face to face. If two or more triangles of a triangle list are adjacent, they appear to have a sharp edge between them.

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

Triangle Interpolants

[This is preliminary documentation and subject to change.]

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

- Color
- Specular
- Alpha

All 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.

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

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

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

Matrices and Transformations

[This is preliminary documentation and subject to change.]

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

- Matrices
- 3-D Transformations

Matrices

[This is preliminary documentation and subject to change.]

Although you needn't be an expert in linear algebra to work with matrices, you should have a passing acquaintance with them. For a refresher in a few of the basics, see 3-D Transformations.

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

[C++]
The D3D_OVERLOADS implementation of the **D3DMATRIX** structure (**D3DMATRIX (D3D_OVERLOADS)**) implements a parentheses ("()") operator. This operator offers convenient access to values in the matrix for C++ programmers. Instead of having to refer to the structure members by name, C++ programmers can refer to them by row and column number, and simply index these numbers as needed. These indices are zero-based, so for example the element in the third row, second column would be M(2, 1). To use the D3D_OVERLOADS operators, you must define D3D_OVERLOADS before including the D3dtypes.h header file.

Also, the D3dutil.cpp source file provides helper functions for creating and concatenating matrices. Feel free to use these functions as they are, or use them as a basis to write your own matrix manipulation functions.

Execute buffer notes

When working with execute buffers, matrices appear to you only as handles. These handles (defined by the **D3DMATRIXHANDLE** type) are used in the D3DOP_MATRIXLOAD and D3DOP_MATRIXMULTIPLY execute buffer opcodes.
You can create a Direct3D matrix by calling the **IDirect3DDevice::CreateMatrix** method. You can set the contents of a matrix by calling the **IDirect3DDevice::SetMatrix** method.

[C++,Visual Basic]

3-D Transformations

[This is preliminary documentation and subject to change.]

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

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

For more information about transformations in Direct3D Immediate Mode, see The Geometry Pipeline.

About 3-D Transformations

[This is preliminary documentation and subject to change.]

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

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

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

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

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

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

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

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

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

[C++]
In C++, Direct3D Immediate Mode declares matrices as a two dimensional array, using the **D3DMATRIX** structure. The following example shows how you would initialize a **D3DMATRIX** structure to act as a uniform scaling matrix:

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

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

```
Dim ScaleMatrix As D3DMATRIX

' In this example, s is a variable of type Single.
With ScaleMatrix
    .rc11 = s
    .rc22 = s
    .rc33 = s
    .rc44 = 1
End With
```

Translation

[This is preliminary documentation and subject to change.]

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

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

You can create a translation matrix by hand in C++, or by using the **Translate** helper function in the D3dutil.cpp file that is included with this SDK. The following example shows the source code for the **Translate** function:

[C++]

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

[Visual Basic]

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

```
Sub TranslateMatrix(m As D3DMATRIX, v As D3DVECTOR)
    Call IdentityMatrix(m)
    m.rc41 = v.x
    m.rc42 = v.y
    m.rc43 = v.z
End Sub
```

Rotation

[This is preliminary documentation and subject to change.]

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

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

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

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

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

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

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

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

[C++]
In a C++ application, you can use the **RotateX**, **RotateY**, and **RotateZ** helper functions in the D3dutil.cpp file to create rotation matrices. (The D3dutil.cpp file is included with this SDK.) The following is the sample code for the **RotateX** helper function:

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

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

[Visual Basic]
Visual Basic application can use the **RotateXMatrix**, **RotateYMatrix**, and **RotateZMatrix** helper subroutines in the Math.bas file to create rotation matrices. (The Math.bas file is included with this SDK.) The following is the sample code for the **RotateXMatrix** helper subroutine:

```
Sub RotateXMatrix(ret As D3DMATRIX, rads As Single)
    Dim cosine As Single
    Dim sine As Single
    cosine = Cos(rads)
    sine = Sin(rads)

    Call IdentityMatrix(ret)

    ret.rc22 = cosine
    ret.rc33 = cosine
    ret.rc23 = -sine
    ret.rc32 = sine
End Sub
```

Scaling

[This is preliminary documentation and subject to change.]

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

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

Matrix Concatenation

[This is preliminary documentation and subject to change.]

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

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

In this formula, *C* is the composite matrix being created, and M_1 through M_n are the individual transformations that matrix *C* will contain. In most cases, you'll only concatenate two or three matrices, but there is no limit.

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

Notice the order in which the matrix multiplication is performed—the order you use is crucial. The preceding formula reflects the right-to-left rule of matrix concatenation. That is, the visible effects of the matrices you use to create a composite matrix occur in right-to-left order. Let's use a typical world transformation matrix as an example. Imagine you were creating the world transformation matrix for a stereotypical "flying saucer." You would probably want to spin the UFO around its center (the y-axis of model space) and translate it to someplace in your scene. To accomplish this effect, you should first create a translation matrix, then multiply it by a rotation matrix, as in the following formula:

$$W = T_w \cdot R_y$$

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

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

No matter what type of matrix you're trying to create, keep the right-to-left rule in mind to ensure that you achieve the expected effects.

Direct3D Immediate Mode Architecture

[This is preliminary documentation and subject to change.]

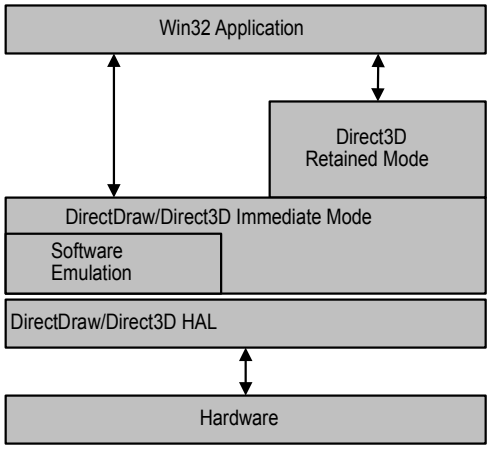
This section provides high-level information about the organization of the Direct3D® Immediate Mode documentation. Information is divided into the following groups:

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

Architectural Overview of Immediate Mode

[This is preliminary documentation and subject to change.]

Direct3D applications communicate with graphics hardware in a similar fashion, whether they use Retained Mode or Immediate Mode. They may or may not take advantage of software emulation before interacting with the HAL. Since Direct3D is an interface to a DirectDraw® object, the HAL is referred to as the DirectDraw/Direct3D HAL.



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

Immediate Mode Object Types

[This is preliminary documentation and subject to change.]

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

[C++]

DirectDraw Object

A DirectDraw object provides the functionality of Direct3D; **IDirect3D, IDirect3D2** and **IDirect3D3** are interfaces to a DirectDraw object. Since a DirectDraw object represents the display device, and the display device implements many of the most important features of Direct3D, it makes sense that the abilities of Direct3D are incorporated into DirectDraw. You create a

DirectDraw object by calling the **DirectDrawCreate** function. For more information, see the Direct3D Interfaces.

DirectDrawSurface Object

A DirectDrawSurface object that was created as a texture map contains the bitmap(s) that your Direct3D application will use as textures. You can retrieve the **IDirect3DTexture2** interface for the surface by calling the **IUnknown::QueryInterface** method, specifying the IID_IDirect3DTexture2 reference identifier.

For more information, see Textures.

Direct3DDevice Object

A Direct3DDevice object encapsulates and stores the rendering state for an Immediate Mode application; it can be thought of as a rendering target for Direct3D. Prior to DirectX 5.0, Direct3D devices were interfaces to DirectDrawSurface objects. DirectX 5.0 introduced a new device-object model, in which a Direct3DDevice object is entirely separate from DirectDraw surfaces. This new object supports the **IDirect3DDevice3** interface.

You can call the **IDirect3D3::CreateDevice** method to create a Direct3DDevice object and retrieve an **IDirect3DDevice3** interface. (Notice that you do not call **QueryInterface** to retrieve **IDirect3DDevice3**.) If necessary, you can retrieve an **IDirect3DDevice** interface by calling the **IDirect3DDevice3::QueryInterface** method.

For more information, see and Direct3D Devices.

Direct3DMaterial Object

A Direct3DMaterial object describes the illumination properties of a visible element in a three-dimensional scene. You can create a Direct3DMaterial object by calling the **IDirect3D3::CreateMaterial** method. You can use the **IDirect3DMaterial3** interface to get and set materials and to retrieve material handles.

For more information, see Materials.

Direct3DViewport Object

A Direct3DViewport object defines the rectangle into which a three-dimensional scene is projected. You can create an **IDirect3DViewport3** interface by calling the **IDirect3D3::CreateViewport** method. For more information, see Viewports and Clipping.

Direct3DLight Object

A Direct3DLight object describes the characteristics of a light in your application. You can use the **IDirect3DLight** interface to get and set lights. You can create an **IDirect3DLight** interface by calling the **IDirect3D3::CreateLight** method.

For more information, see Lights.

Direct3DVertexBuffer Object

A Direct3DVertexBuffer object is a memory buffer that contains vertices to be rendered with the vertex-buffer rendering methods offered in the **IDirect3DDevice3** interface. Vertex buffers are not to be confused with execute buffers. A vertex buffer contains only vertex data to be processed by a

Direct3DDevice object, and offers features that you can use to improve performance during rendering.

For additional information, see Vertex Buffers.

Direct3DExecuteBuffer Object

A Direct3DExecuteBuffer object is a buffer full of vertices and instructions about how to handle them. Prior to DirectX 5.0, Immediate Mode programming was done exclusively by using Direct3DExecuteBuffer objects. The introduction of the DrawPrimitive methods in DirectX 5.0, however, has made it unnecessary for most applications to work with execute buffers.

For more information about execute buffers, see Execute Buffers.

[Visual Basic]

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

DirectX7 Class

The **DirectX7** class defines the object that is used to create all other component-level objects like the **Direct3D3** or **DirectDraw4** object.

DirectDraw4 Class

The **DirectDraw4** class represents the display device, spawns the **Direct3D3** class, and creates the texture and rendering target surfaces used for rendering. You create an object of the **DirectDraw4** class by calling the **DirectX7.DirectDrawCreate** method.

DirectDrawSurface4 Class

The **DirectDrawSurface4** class represents memory that is used to contain image data for display, texturing, or as a rendering target for a Direct3D device. Call the **DirectDraw4.CreateSurface** method to create an empty surface, or call the **DirectDraw4.CreateSurfaceFromFile** method to create a surface and load it with image data from a bitmap file.

Direct3D3 Class

The **Direct3D3** class defines an object that provides the functionality of Direct3D, used to create and manipulate the objects you will need to render a scene. You create an object of the **Direct3D3** class by calling the **DirectDraw4.GetDirect3D** method.

Direct3DDevice3 Class

The **Direct3DDevice3** class defines an object that encapsulates and stores the rendering state for an Immediate Mode application written in Visual Basic. You create an object of this class by calling the **Direct3D3.CreateDevice** method.

Direct3DEnumDevices Class

The **Direct3DEnumDevices** class defines an object that can enumerate the rendering devices present on a system. You create an object of the **Direct3DEnumDevices** class by calling the **Direct3D3.GetDevicesEnum** method.

Direct3DEnumPixelFormats Class

The **Direct3DEnumPixelFormats** class defines an object that can enumerate the pixel formats supported for a given device on a system. To create an object of the **Direct3DEnumPixelFormats** class, call the **Direct3D3.GetEnumZBufferFormats** or **Direct3DDevice3.GetTextureFormatsEnum** methods.

Direct3DLight Class

The **Direct3DLight** class defines an object that describes the characteristics of a light in a 3-D scene application. You can create lights by calling the **Direct3D3.CreateLight** method.

For more information, see Lights

Direct3DMaterial3 Class

The **Direct3DMaterial3** class defines an object that describes the illumination properties of a visible element in a three-dimensional scene, called a material. You can create a material by calling the **Direct3D3.CreateMaterial** method.

For more information, see Materials.

Direct3DTexture2 Class

The **Direct3DTexture2** class defines a DirectDraw surface object that contains image data to be rendered onto the faces of geometry. The **DirectDrawSurface4.GetTexture** method creates a texture class for a surface that was created with the DDSCAPS_TEXTURE capability flag.

For more information, see Textures.

Direct3DVertexBuffer Class

The **Direct3DVertexBuffer** class defines an object that acts as a memory buffer for vertices. Vertices in a vertex buffer can be rendered with the vertex-buffer rendering methods offered in the **Direct3DDevice3** class. Call the **Direct3D3.CreateVertexBuffer** method to create a vertex buffer.

For additional information, see Vertex Buffers.

Direct3DViewport3 Class

The **Direct3DViewport3** class defines an object that describes the rectangle into which a three-dimensional scene is projected. You can create a viewport by calling the **Direct3D3.CreateViewport** method.

For more information, see Viewports and Clipping.

Immediate Mode COM Interfaces

[This is preliminary documentation and subject to change.]

[Visual Basic]

Note

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

[C++]
The Direct3D Immediate Mode API consists primarily of the following COM interfaces:

IDirect3D3	Root interface, used to obtain other interfaces
IDirect3DDevice	3D Device for execute-buffer based programming
IDirect3DDevice3	3D Device for DrawPrimitive-based programming
IDirect3DLight	Interface used to work with lights
IDirect3DMaterial3	Surface-material interface
IDirect3DTexture2	Texture-map interface
IDirect3DVertexBuffer	Interface used to work with vertex buffers.
IDirect3DViewport3	Interface to define the viewport's characteristics.
IDirect3DExecuteBuffer	Interface for working with execute buffers

For backward compatibility with previous versions of DirectX, all former interface versions are still supported in DirectX 6.0.

DrawPrimitive Methods and Execute Buffers

[This is preliminary documentation and subject to change.]

[Visual Basic]

Note
The information in this topic applies only to applications written in C++. DirectX for Visual Basic does use execute buffer rendering.

[C++]
DirectX 5.0 introduced a radically new way to use Direct3D Immediate Mode. Previously, you had to fill and execute the execute buffers to accomplish any task. Now, you can use the DrawPrimitive methods, which allow you to draw primitives directly.

The **IDirect3DDevice** interface supports execute buffers. The **IDirect3DDevice3** interface supports the DrawPrimitive methods. Despite the names of these interfaces, **IDirect3DDevice3** is not a COM iteration of the **IDirect3DDevice** interface. Although there is some overlap in the functionality of the interfaces, they are separate implementations. This means that you cannot call **IDirect3DDevice::QueryInterface** to retrieve an **IDirect3DDevice3** interface. You must call the **IDirect3D3::CreateDevice** method, instead.

For more information about working with execute buffers, see Using Execute Buffers.

For more information about the DrawPrimitive methods, see Rendering. For more information about device objects, see Direct3D Devices.

Direct3D Immediate Mode Essentials

[This is preliminary documentation and subject to change.]

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

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

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

Immediate Mode Changes for DirectX 7.0

[This is preliminary documentation and subject to change.]

DirectX 7.0 is backward compatible with all prior versions of DirectX. Direct3D—like all of DirectX 7.0—now supports Visual Basic applications. Direct3D for DirectX 7.0 incorporates new functionality to improve performance, increase ease-of-use, and exploit new hardware features.

Direct3D and DirectDraw

[This is preliminary documentation and subject to change.]

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

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

The DirectDraw Object and Direct3D

[This is preliminary documentation and subject to change.]

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

[C++]
The Direct3D interfaces are actually interfaces to the DirectDraw object. DirectDraw presents programmers with a single, unified object that encapsulates both the DirectDraw and Direct3D states. The DirectDraw object is the first object your application creates (by calling **DirectDrawCreate**) and the last object your application releases. Since the DirectDraw object represents the display device, and the display device implements many of the most important features of Direct3D, it makes sense that the abilities of Direct3D are incorporated into DirectDraw.

The important implication of this is that the lifetime of the Direct3D driver state is the same as that of the DirectDraw object. Releasing the Direct3D interface does not destroy the Direct3D driver state. That state is not destroyed until all references to that object — whether they are DirectDraw or Direct3D references — have been released. Therefore, if you release a Direct3D interface while holding a reference to a DirectDraw driver interface, and then query the Direct3D interface again, the Direct3D state will be preserved.

The DirectDraw object contains three Direct3D interfaces: **IDirect3D, IDirect3D2**, and **IDirect3D3**. The **IDirect3D** and **IDirect3D2** interfaces are obsolete and are provided for backward compatibility. New applications should use the **IDirect3D3** interface to create other Direct3D objects such as viewports, lights, textures, and materials. For details, see Direct3D Interfaces.

[Visual Basic]
Although the DirectDraw and Direct3D share much of the same code — if fact, deep down, they're the same root object — DirectX for Visual Basic exposes them according to how you will use them: as functionally separate classes. DirectX for Visual Basic uses the **DirectDraw4** and **Direct3D3** classes from which your application can create objects. The underlying relationship (as described in the C++ portion of this topic) between DirectDraw and Direct3D is apparent even from within Visual Basic. Notice that the global **DirectX7** class contains a method to create a **DirectDraw4** object, but no equivalent method for Direct3D. Rather, the **DirectDraw4** class defines a method to "retrieve" the **Direct3D3** class, called **DirectDraw4.GetDirect3D**. This because these two classes actually refer to the same object within DirectX!

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

The **DirectDraw4** class provides the two dimensional features for a display device, while the **Direct3D3** class offers the 3-D features for the device. You use the **DirectDraw4** class to create the surfaces that the Direct3D class will use for rendering targets, depth buffers, and textures. The Direct3D class is then used to create the subordinate objects that Direct3D requires, such as viewports, lights, and materials.

Direct3D Interfaces

[This is preliminary documentation and subject to change.]

[Visual Basic]

Note

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

[C++]
The functionality of Direct3D objects is accessed by C++ applications through the interfaces **IDirect3D, IDirect3D2** and **IDirect3D3**. This section presents information on all three of these interfaces.

IDirect3D Interface

The **IDirect3D** interface supports the use of execute buffers. It is provided for compatibility with existing code. New applications should use the **IDirect3D3** interface (see Direct3D Interfaces).

If your application uses execute buffers, it must obtain a pointer to the **IDirect3D** interface from a DirectDraw object using the **QueryInterface** method.

IDirect3D2 Interface

A more straightforward style of Direct3D programming was introduced with the **IDirect3D2** interface. This programming style is based on rendering primitives with a single call rather than rendering through an execute buffer. Devices exposed two methods for this new rendering style, **IDirect3DDevice2::DrawPrimitive** and **IDirect3DDevice2::DrawIndexedPrimitive** methods; these methods are supported in the most recent device interface, **IDirect3DDevice3**.

Applications get a pointer to the **IDirect3D2** interface from a DirectDraw object using the **QueryInterface** method.

The **IDirect3D2** interface is the starting point for creating other Direct3D Immediate Mode interfaces. Using it, your application can find and enumerate the types of Direct3D devices supported by a particular DirectDraw object. It also has methods needed to create other Direct3D Immediate Mode objects, such as viewports, materials and lights.

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

Note also that You cannot use new features that the **IDirect3DDevice2** interface offers with **IDirect3DDevice** interface pointers. If your application calls the **QueryInterface** method on a DirectDraw surface and retrieves an **IDirect3DDevice**, it cannot access the features supported by later device interfaces. In addition, it cannot call the **QueryInterface** method on a device object created in this way to retrieve an **IDirect3DDevice2** interface.

IDirect3D3 Interface

Like the **IDirect3D2** interface, the **IDirect3D3** interface supports the DrawPrimitive style of Direct3D programming. In addition, the **IDirect3D3** interface extends the functionality of the DrawPrimitive-style programming by introducing the **IDirect3DDevice3** interface. This new interface supports flexible vertex formats, vertex buffers, and new texturing capabilities. For additional information, see Vertex Formats, Textures, Vertex Buffers and Rendering.

Applications obtain a pointer to the **IDirect3D3** interface from a DirectDraw object using the **QueryInterface** method. For details, see Accessing Direct3D.

Accessing Direct3D

[This is preliminary documentation and subject to change.]

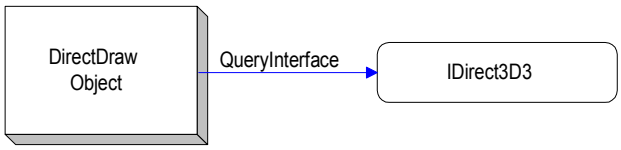
[C++]

When a Direct3D application written in C++ starts up, it must obtain a pointer to an **IDirect3D3** interface to access Direct3D functionality. Use the following steps to obtain a pointer to the **IDirect3D3** interface:

To obtain a pointer to the IDirect3D3 interface

1. Call **DirectDrawCreate** to create a DirectDraw device.
2. Call the **IUnknown::QueryInterface** method to get a pointer to an **IDirect3D3** interface.

The following diagram illustrates these steps:



The following code fragment demonstrates how to query for an **IDirect3D3** interface:

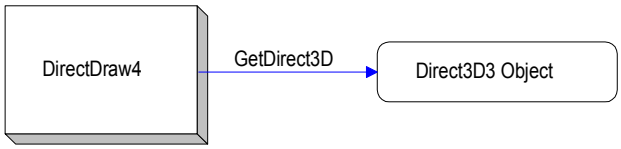
```
LPDIRECTDRAW    lpDD;           // IDirectDraw Interface
LPDIRECT3D3     lpD3D;          // IDirect3D3 Interface

// Get an IDirectDraw interface.
// Use the current display driver.
hResult = DirectDrawCreate (NULL, &lpDD, NULL);
if (FAILED (hResult))
{
   // Code to handle an error goes here.
}

// Get D3D interface
hResult =
   lpDD->QueryInterface (IID_IDirect3D3, (void **)&lpD3D);
if (FAILED (hResult))
{
   // Code to handle the error goes here.
}
```

[Visual Basic]
When a Direct3D application written in Visual Basic begins, it must obtain a reference to the **Direct3D3** class to access Direct3D functionality. Use the following steps to obtain such a reference:

Û To obtain a reference to the Direct3D3 class

1. Call **DirectX7.DirectDrawCreate** to create a **DirectDraw4** class object.
2. Call **DirectDraw4.GetDirect3D** method to retrieve the DirectDraw object's reference to the **Direct3D3** class.

The following diagram illustrates these steps:



The following code fragment demonstrates how to retrieve the **Direct3D3** class from a **DirectDraw4** class object:

```
' Retrieve a reference to the Direct3D3 class.
'
' For this example, dx is a valid reference to the DirectX7 class.
On Local Error Resume Next
Dim ddraw As DirectDraw4
Dim d3d As Direct3D3

' Create a default DirectDraw4 object.
Set ddraw = dx.DirectDrawCreate("")
If (Err.Number <> DD_OK) Then
   ' Handle error.
End If

' Retrieve the Direct3D3 reference from the DirectDraw4 object.
Set d3d = ddraw.GetDirect3D
If (Err.Number <> DD_OK) Then
   ' Handle error.
End If
```

Creating Objects Subordinate to Direct3D

[This is preliminary documentation and subject to change.]

[C++]
Applications written in C++ use the **IDirect3D3** interface to create Direct3D devices, viewports, lights, materials, and textures. Textures under the **IDirect3D2** interface were manipulated using texture handles. The **IDirect3D3** interface significantly extends the texturing capabilities of Direct3D. To accommodate these changes, the **IDirect3D3** interface no longer utilizes texture handles. Instead, it makes use of texture interfaces. For more information, see Textures.

Direct3D creates devices by invoking the **IDirect3D3::CreateDevice** method. For details, see Direct3D Devices.

The **IDirect3D3::CreateViewport** method is called when creating viewports. For more information, see Viewports and Clipping.

Lights and materials are created through calls to the **IDirect3D3::CreateLight** and **IDirect3D3::CreateMaterial** methods respectively. For more information, see Lighting and Materials.

[Visual Basic]
Applications written in Visual Basic use the methods of the **Direct3D3** class to create Direct3D devices, viewports, lights, and materials.

Direct3D creates devices by invoking the **Direct3D3.CreateDevice** method. For details, see Direct3D Devices.

The **Direct3D3.CreateViewport** method is called when creating viewports. For more information, see Viewports and Clipping.

Lights and materials are created through calls to the **Direct3D3.CreateLight** and **Direct3D3.CreateMaterial** methods respectively. For more information, see Lighting and Materials.

DirectDraw Cooperative Levels and FPU Precision

[This is preliminary documentation and subject to change.]

[Visual Basic]

Note
The information in this topic applies only to applications written in C++. DirectX for Visual Basic does not support the DDSCL_FPUSETUP flag.

[C++]
Direct3D always uses single-precision floating point calculations to increase the performance of rendering a 3-D scene. By default, Direct3D checks the precision state of the FPU (usually it's set for double precision), sets it to single precision, performs the necessary operations, then sets the FPU back to double precision before returning control to the calling application. This process is repeated for each rendering cycle.

You can improve the performance of Direct3D by including the DDSCL_FPUSETUP cooperative level flag when setting the DirectDraw cooperative level. Basically, this flag informs the system that your application does not rely on the FPU being set to double-precision. When you use DDSCL_FPUSETUP, Direct3D sets the FPU to single precision once, reinstating double-precision FPU calculations only when Direct3D is shut down, saving considerable performance overhead by eliminating the process of setting and resetting the FPU for each rendering cycle.

Obviously, if you set DDSCL_FPUSETUP, your application will be subject to the limitations of single-precision floating point values. As a result, this cooperative level

setting should only be used when single-precision floating point values are acceptable for your application. If you require some double-precision floating point calculations, you can manually set the FPU precision mode to double as needed, but you must reset it to single precision before calling any Direct3D method. Failing to reset the FPU to single-precision mode when DDSCL_FPUSETUP will result in degraded performance.

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

Important

Loading some dynamic link libraries (DLLs) at runtime can cause the FPU to be reset to double-precision mode. Some compilers, such as Microsoft Visual C++, set the default DLL entry point to **_DllMainCrtStartup**, a function that the compiler supplies to initialize the C/C++ runtime components. This function also sets the FPU precision mode to double precision. If an application sets the DDSCL_FPUSETUP cooperative level, then loads a DLL, Direct3D will not detect that the FPU has been reset, and performance will suffer.
If your application loads DLLs at runtime, it should check and reset the FPU precision mode immediately after the **LoadLibrary** Win32 function returns, and before calling any Direct3D functions. You can reset the precision mode explicitly or by calling the **IDirectDraw4::SetCooperativeLevel** method again, including the DDSCL_FPUSETUP flag. Using **SetCooperativeLevel** to set the FPU precision mode can also cause DirectDraw surfaces to be lost.
(You can explicitly set the entry point for DLLs you compile by using the /ENTRY: linker switch, but if you do, the C/C++ runtime will not be initialized automatically.)

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

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

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

   __asm fstcw wSave
   if (wSave & 0x300 ||          // Not single mode
```

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

void RestoreFPCW(WORD wSave)
{
    __asm fldcw wSave
}

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

Direct3D Devices

[This is preliminary documentation and subject to change.]

This section provides an overview of the purpose for and uses of Direct3D devices. The overview is divided the following topics:

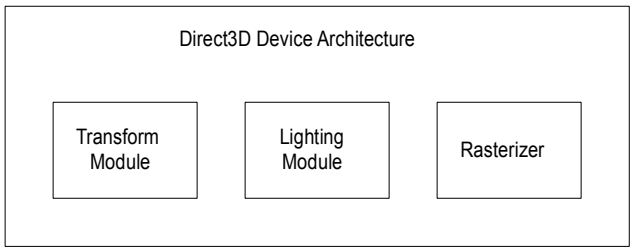
- What Is a Direct3D Device?
- Direct3D Device Types
- Device Interfaces
- Using Devices
- Emulation Modes

- AGP Surfaces and Direct3D Devices
- Execute Buffers

What Is a Direct3D Device?

[This is preliminary documentation and subject to change.]

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



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

[C++]
In C++, Direct3D Immediate Mode devices support three device interfaces: **IDirect3DDevice**, **IDirect3DDevice2**, and **IDirect3DDevice3**. The **IDirect3DDevice** interface provides methods used for programming with execute buffers. However, this is a form of Direct3D development that is provided primarily for backward compatibility. The **IDirect3DDevice2** interface introduced the DrawPrimitive methods of Direct3D programming. The **IDirect3DDevice3** extends the DrawPrimitive functionality. The DrawPrimitive methods represent the preferred rendering approach. The interfaces share a few common methods that are useful in either programming style, and these methods are provided by all three interfaces for your convenience. For more information about the two rendering approaches, see Rendering.

Prior to the introduction of the **IDirect3DDevice2** interface in DirectX 5.0, Direct3D devices were interfaces to DirectDrawSurface objects. The **IDirect3DDevice2** interface implements a device-object model in which a Direct3DDevice object is entirely separate from DirectDraw surfaces. The **IDirect3DDevice3** interface uses and extends the same device-object model. Because they are separated from DirectDraw surfaces and have independent lifetimes, Direct3D device objects can use different DirectDraw surfaces as render targets at different times, if the application requires it. For information about rendering targets, see **IDirect3DDevice3::SetRenderTarget**.

[Visual Basic]
In DirectX for Visual Basic, you access Direct3D Immediate Mode devices through the **Direct3DDevice3** class. The **Direct3DDevice3** class supports the DrawPrimitive family of scene-rendering methods. The **Direct3DDevice3** object can use different DirectDraw surfaces as render targets at different times, if the application requires it. For information about rendering targets, see **Direct3DDevice3.SetRenderTarget**.

Direct3D Device Types

[This is preliminary documentation and subject to change.]

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

- About Device Types
- HAL Device
- RGB Device
- Reference Rasterizer
- Legacy Device Types

About Device Types

[This is preliminary documentation and subject to change.]

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

[C++]

Note

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

The Direct3D device that an application creates must correspond to the capabilities of the hardware on which the application is running. Direct3D provides rendering capabilities either by accessing 3-D hardware that is installed in the computer, or by emulating the capabilities of 3-D hardware in software. Therefore, Direct3D provides devices for both hardware access and for software emulation.

Hardware-accelerated devices give better performance than software-emulated devices. In most cases, your application will target machines that have hardware

acceleration of some kind, and fall back on software emulation to accommodate lower-end computers.

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

[C++,Visual Basic]

HAL Device

[This is preliminary documentation and subject to change.]

If the computer on which your application is running is equipped with a display adapter that supports Direct3D, your application should use it for 3-D operations. Direct3D HAL devices implement all or part of the transformation, lighting, and rasterizing modules in hardware.

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

[C++]
To create a HAL device from C++, call the **IDirect3D3::CreateDevice** method and pass the IID_IDirect3DHALDevice value as the first parameter. For details, see Creating a Direct3D Device.

[Visual Basic]
To create a HAL device from Visual Basic, call the **Direct3D3.CreateDevice**, and pass the "IID_IDirect3DHALDevice" string constant as the first parameter. For details, see Creating a Direct3D Device.

Note

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

RGB Device

[This is preliminary documentation and subject to change.]

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

some AMD processors, and the MMX instruction set supported by many Intel processors. Supported instruction sets include the AMD 3D-Now! instruction set on some AMD processors, and the MMX instruction set supported by many Intel processors. Direct3D utilizes the 3D-Now! instruction set to accelerate transformation and lighting operations, and the MMX instruction set to accelerate rasterization.

For more information, see Emulation Modes.

[C++]
Applications written in C++ create an RGB device with the **IDirect3D3::CreateDevice** method. Pass the value IID_IDirect3DRGBDevice as the first parameter. For additional information on creating Direct3D devices, see Creating a Direct3D Device.

[Visual Basic]
Applications written in Visual Basic create an RGB device with the **Direct3D3.CreateDevice** method. Pass the "IID_IDirect3DRGBDevice" string constant as the first parameter. For additional information on creating Direct3D devices, see Creating a Direct3D Device.

Reference Rasterizer

[This is preliminary documentation and subject to change.]

Direct3D supports an additional device type called the reference rasterizer. Unlike an RGB device, the reference rasterizer supports every Direct3D feature. Because these features are implemented for accuracy in favor of speed, and are implemented in software, the results aren't very fast. The reference rasterizer does make use of special CPU instructions whenever it can, but it is not intended for retail applications. You should only use the reference rasterizer for feature testing or demonstration purposes.

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

[C++]
Applications written in C++ create a reference device with the **IDirect3D3::CreateDevice** function. Pass the value IID_IDirect3DRefDevice as the first parameter. For additional information on creating Direct3D devices, see Creating a Direct3D Device.

[Visual Basic]
Visual Basic applications create a reference device with the **Direct3D3.CreateDevice** function. Pass the "IID_IDirect3DRefDevice" string constant as the first parameter. For additional information on creating Direct3D devices, see Creating a Direct3D Device.

Legacy Device Types

[This is preliminary documentation and subject to change.]

[Visual Basic]

Note

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

[C++]

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

MMX Device

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

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

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

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

Ramp Device

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

A ramp device is a software-emulated device that provides monochrome lighting. Applications can select it when the user's computer does not have sufficient

processing power to support any other types of Direct3D devices. For details, see Emulation Modes.

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

An application creates a ramp device by invoking the **IDirect3D2::CreateDevice** method and passing the value IID_IDirect3DRampDevice as the first parameter. For additional information on creating Direct3D devices, see Creating a Direct3D Device.

Device Interfaces

[This is preliminary documentation and subject to change.]

[Visual Basic]

Note

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

[C++]

Applications written in C++, use a device interface to manipulate a Direct3DDevice object's rendering states, lighting states, and to perform rendering operations. Although devices support three iterations of the device interface (**IDirect3DDevice**, **IDirect3DDevice2**, and **IDirect3DDevice3**), it is unlikely that your applications will need to use more than one of them at a time. Which interface you use should be determined by the rendering approach — the DrawPrimitive methods or execute buffers — your application is going to use. The following paragraphs provide additional information about these interfaces and the rendering approaches they represent:

Execute buffers

The **IDirect3DDevice** interface supports rendering through execute buffers; the original method of Direct3D programming. This interface is supported mostly for backward compatibility, as the DrawPrimitive rendering architecture is easier to use. It is recommended that you use the **IDirect3DDevice3** interface for developing new applications, unless your application specifically targets versions of DirectX prior to DirectX 5.0.

All Direct3D device types support the **IDirect3DDevice** interface.

DrawPrimitive rendering

The two most recent iterations of the device interface, **IDirect3DDevice3** and **IDirect3DDevice2**, support the DrawPrimitive family of rendering methods in favor

of rendering with execute buffers. The DrawPrimitive methods greatly simplify the process of preparing and rendering vertices to the render target surface, and are widely considered the preferred rendering approach.

If your application uses the DrawPrimitive methods, there is little reason to use anything but the newest version of the device interface (in fact, this documentation provides information only for the most recent version). Direct3D, in compliance with COM standards for backward compatibility, supports all interface versions, but it is recommended that you use the most recent version to take advantage of any new features and performance enhancements.

The **IDirect3DDevice2** interface, created for DirectX 5.0, introduced the DrawPrimitive rendering architecture. DrawPrimitive-based rendering methods are simpler to use than execute buffers and provide a more direct style of Direct3D programming. For details, see Rendering. All Direct3D device types support the **IDirect3DDevice2** interface.

Like the **IDirect3DDevice2** interface, the **IDirect3DDevice3** interface provides support for the DrawPrimitive methods. In conjunction with the **IDirect3D3** interface, it also sports an enhanced feature set which includes multiple texture blending, vertex buffers, and enhancements to the 3-D geometry rendering pipeline. The **IDirect3DDevice3** interface furnishes the same methods that the **IDirect3DDevice2** interface did. However, it adds new render states to the **IDirect3DDevice3::SetRenderState** method to support multiple texture blending.

All Direct3D device types support the **IDirect3DDevice3** interface.

Using Devices

[This is preliminary documentation and subject to change.]

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

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

Enumerating Direct3D Devices

[This is preliminary documentation and subject to change.]

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

- Starting Device Enumeration

- Selecting an Enumerated Device

Starting Device Enumeration

[This is preliminary documentation and subject to change.]

[C++]
A C++ application begins device enumeration by calling the **IDirect3D3::EnumDevices** method, which enumerates all of the Direct3D devices that the hardware supports. Direct3D invokes the **D3DEnumDevicesCallback** function to select the device that will be used. The **D3DEnumDevicesCallback** function is supplied by you in your application. Note that because this callback function is supplied by your application, its name can be anything you want.

The following code fragment illustrates the process of enumerating Direct3D devices. In this example, the device enumeration callback function is named EnumDeviceCallback. The code passes a pointer to EnumDeviceCallback to the **IDirect3D3::EnumDevices** method, which in turn calls EnumDeviceCallback for each device being enumerated, then returns. The callback can stop the enumeration before all devices have been enumerated by returning D3DENUMRET_CANCEL.

```
// In this code fragment, the variable lpd3d contains a valid
// pointer to the IDirect3D3 interface that the application obtained
// prior to executing this code.

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

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

[Visual Basic]
A Visual Basic application begins device enumeration by retrieving a reference to the **Direct3DEnumDevices** class by calling the **Direct3D3.GetDevicesEnum** method. The **Direct3DEnumDevices** class defines methods that retrieve the number of enumerated devices and information about each device.

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

```
' In this example, the d3d variable is a valid reference to a
' Direct3D3 class.
```

```
Dim d3dEnum As Direct3DEnumDevices

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

Selecting An Enumerated Device

[This is preliminary documentation and subject to change.]

[C++]
In C++, Direct3D invokes the **D3DEnumDevicesCallback** function provided by the caller for each Direct3D device installed on the system. When it is called, its first parameter is a globally unique ID (GUID) for the device that is being enumerated. The value of the GUID will be IID_IDirect3DHALDevice, IID_IDirect3DMMXDevice, IID_IDirect3DRGBDevice, or IID_IDirect3DRampDevice. The **D3DEnumDevicesCallback** function selects the device that is most appropriate for your application based on this information.

The second and third parameters to the **D3DEnumDevicesCallback** function are text strings containing the name and user-friendly description of the device.

The fourth parameter is a pointer to a **D3DDEVICEDESC** structure containing information about the hardware capabilities of the device. Even if the device being enumerated is a HAL device, the particular hardware may not support all of the capabilities that the Direct3D API allows.

The fifth parameter to the **D3DEnumDevicesCallback** function contains a pointer to a **D3DDEVICEDESC** structure that describes the software-emulated capabilities of the computer on which your application is running. This information is relevant when you are using a software-emulated device (MMX, RGB, or RAMP device).

The last parameter is a programmer-defined value. Your application passes this value to the **IDirect3D3::EnumDevices** method. It in turn passes this value to the **D3DEnumDevicesCallback** function.

The following code fragment illustrates how to create a **D3DEnumDevicesCallback** function. In this example, the application-supplied callback function is named EnumDeviceCallback. The EnumDeviceCallback function uses the following algorithm to choose an appropriate Direct3D device:

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

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

```
// This function is written with the assumption that the following
// global variables are declared in the program.
```

```
// DWORD                dwDeviceBitDepth        = 0;
// GUID                 guidDevice;
// char                 szDeviceName[MAX_DEVICE_NAME];
// char                 szDeviceDesc[MAX_DEVICE_DESC];
// D3DDEVICEDESC        d3dHWDeviceDesc;
// D3DDEVICEDESC        d3dSWDeviceDesc;

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

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

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

   // Does the device render at the depth we want?
    if ((lpd3dDeviceDesc->dwDeviceRenderBitDepth & dwDeviceBitDepth)
      == 0)
   {
      // If not, skip this device.

      return D3DENUMRET_OK;
   }

   // The device must support Gouraud-shaded triangles.

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

         return D3DENUMRET_OK;
      }
   }
   else
```

```
    {
        if (!(lpd3dDeviceDesc->dpcTriCaps.dwShadeCaps &
              D3DPSHADECAPS_COLORGOURAUDRGB))
        {
            // No Gouraud shading. Skip this device.

            return D3DENUMRET_OK;
        }
    }

    //
    // This is a device we are interested in. Save the details.
    //
    *lpUserArg = TRUE;
    CopyMemory(&guidDevice, lpGUID, sizeof(GUID));
    strcpy(szDeviceDesc, lpszDeviceDesc);
    strcpy(szDeviceName, lpszDeviceName);
    CopyMemory(&d3dHWDeviceDesc, lpd3dHWDeviceDesc,
               sizeof(D3DDEVICEDESC));
    CopyMemory(&d3dSWDeviceDesc, lpd3dSWDeviceDesc,
               sizeof(D3DDEVICEDESC));

    // If this is a hardware device, we have found
    // what we are looking for.
     if (fIsHardware)
        return D3DENUMRET_CANCEL;

    // Otherwise, keep looking.

    return D3DENUMRET_OK;
}
```

[Visual Basic]

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

Call the **Direct3DEnumDevices.GetGuid** method retrieves the globally-unique ID (GUID) for an enumerated device as a text string. The string differs from one device to another.

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

Call the **Direct3DEnumDevices.GetHWDesc** and **Direct3DEnumDevices.GetHELDesc** methods to retrieve a description of the hardware and software capabilities for the device, in the form of the **D3DDEVICEDESC** type. Even if the device being enumerated is a hardware device, the particular hardware may not support all of the capabilities that the Direct3D API allows.

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

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

```
Private Sub EnumerateDevices(d3denum As Direct3DEnumDevices)
' For this example, the following global variables are assumed to be defined:
   '  g_lDeviceBitDepth is the desired bit depth (combination of CONST_DDBITDEPTHFLAGS
constants).
   '  g_HELDeviceDesc will contain the D3DDEVICEDESC for software device capabilities.
   '  g_HWDeviceDesc will contain the D3DDEVICEDESC for hardware device capabilities.
   '  g_strDeviceGUID will contain the GUID string for the device.
   '  g_strDeviceDescription will contain the string for the device description text.
   '  g_strDeviceName will contain the string for the device name.

   ' Local variables used to enumerate devices.
   Dim iDevice As Integer
   Dim HWDevDesc As D3DDEVICEDESC, SWDevDesc As D3DDEVICEDESC
   Dim bIsHardware As Boolean

   ' Begin enumerating the devices
   For iDevice = 1 To d3denum.GetCount
      Dim checkDesc As D3DDEVICEDESC
      Call d3denum.GetHWDesc(iDevice, HWDevDesc)
      Call d3denum.GetHELDesc(iDevice, SWDevDesc)

      ' If there is no hardware support for this device, the color model is zero.
      bIsHardware = (HWDevDesc.lColorModel <> 0)
      If bIsHardware = True Then
         checkDesc = HWDevDesc
      Else
         checkDesc = SWDevDesc
```

```
        End If

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

        ' The device must support Gouraud-shaded triangles.
        If checkDesc.lColorModel = D3DCOLOR_MONO Then
            If (Not checkDesc.dpcTriCaps.lShadeCaps And
D3DPSHADECAPS_COLORGOURAUDMONO) Then
                ' No Gouraud shading in MONO mode. Skip this device.
                GoTo NEXT_DEVICE
            End If
        ElseIf (Not checkDesc.dpcTriCaps.lShadeCaps And
D3DPSHADECAPS_COLORGOURAUDRGB) Then
            ' No Gouraud shading in RGB mode. Skip this device.
            GoTo NEXT_DEVICE
        End If

        '
        ' By the time we get here, we know that this is a device we
        ' are interested in. Place the results in the global variables.
        '
        g_strDeviceGUID = d3denum.GetGuid(iDevice)
        g_strDeviceDescription = d3denum.GetDescription(iDevice)
        g_strDeviceName = d3denum.GetName(iDevice)
        g_SWDeviceDesc = SWDevDesc
        g_HWDeviceDesc = HWDevDesc

        ' If this current device happens to be hardware accelerated,
        ' we have found what we are looking for.
        If bIsHardware = True Then Exit For

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

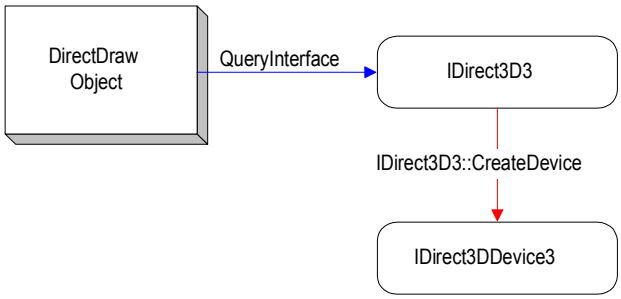
Creating a Direct3D Device

[This is preliminary documentation and subject to change.]

[C++]

To create a Direct3D device in a C++ application, your application must first initialize the DirectDraw object in the normal manner and obtain a pointer to the **IDirect3D3** interface. For details, see Accessing Direct3D. It must then call the **IDirect3D3::CreateDevice** method to create a Direct3D device. This method will pass a pointer to the **IDirect3DDevice3** interface your application.

The following figure illustrates the process for creating a Direct3D device in C++. Such a device would support the DrawPrimitive methods. For information about creating a device for use with execute-buffer rendering, see Creating a Device for Execute Buffers.



This code fragment illustrates this process:

```
LPDIRECTDRAW    lpDD;              // DirectDraw Interface
LPDIRECT3D3     lpD3D;             // Direct3D3 Interface
LPDIRECTDRAWSURFACE4  lpddsRender;  // Rendering surface
LPDIRECT3DDEVICE3     lpd3dDevice;  // D3D Device

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

// Get an IDirect3D3 interface
hResult =
   lpDD->QueryInterface (IID_IDirect3D3, (void **)&lpD3D);
if (FAILED (hResult))
{
   // Code to handle the error goes here.
}

//
// Code for the following tasks is omitted for clarity.
//
// Applications will need to set the cooperative level at this point.
```

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

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

The preceding sample invokes the **IDirect3D3::CreateDevice** method to create the Direct3D device. In the case of this sample, a Direct3D HAL device is created if the call is successful.

Note that the target DirectDraw rendering surface that your application creates must be created for use as a Direct3D rendering target. To do this, it must pass a **DDSURFACEDESC2** structure to the **IDirectDraw4::CreateSurface** method. The **DDSURFACEDESC2** structure has a member called **ddsCaps**, which is a structure of type **DDSCAPS**. The **DDSCAPS** structure contains a member named **dwCaps**, which must be set to DDSCAPS_3DDEVICE when **IDirectDraw4::CreateSurface** is invoked.

The render target surface you use must be created in display memory (with the DDSCAPS_VIDEOMEMORY flag) when it will be use with a hardware-accelerated rendering device, and in system memory (using the DDSCAPS_SYSTEMMEMORY flag) otherwise.

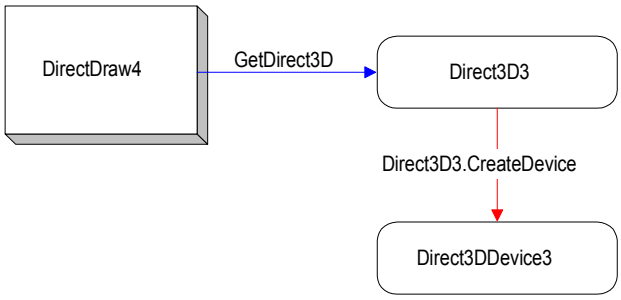
Note

Some popular hardware devices require that the render target and depth buffer surfaces use the same bit depth. On such hardware, if your application uses a 16-bit render target surface, the attached depth buffer must also be 16-bits. For a 32-bit render target surface, the depth buffer must be 32-bits, of which 8-bits can be used for stencil buffering (if needed).
If the hardware upon which your application is running has this requirement, and your application fails to meet it, any attempts to create a rendering device that uses the non-compliant surfaces will fail. You can use the DirectDraw method, **IDirectDraw4::GetDeviceIdentifier** to track hardware that imposes this limitation.

[Visual Basic]
To create a Direct3D device, your application must first initialize the DirectDraw object in the normal manner and obtain a reference to the **Direct3D3** class. For details, see Accessing Direct3D. It must then call the **Direct3D3.CreateDevice**

method to create a Direct3D device. The method returns a reference to a **Direct3DDevice3** class object.

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



This code fragment illustrates this process:

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

    Dim ddraw As DirectDraw4        ' DirectDraw4 class
    Dim d3d As Direct3D3            ' Direct3D3 class
    Dim ddsRender As DirectDrawSurface4 ' Rendering surface
    Dim d3dDev As Direct3DDevice3   '  Direct3D Device

    ' Create DirectDraw4 object.
    ' Use the current display driver.
    Set ddraw = g_dx.DirectDrawCreate("")
    If Err.Number <> DD_OK Then
        ' Code to handle the error goes here.
    End If

    ' Get a Direct3D3 object
    Set d3d = ddraw.GetDirect3D
    If Err.Number <> DD_OK Then
        ' Code to handle the error goes here.
    End If

    '
    ' Code for the following tasks is omitted for clarity.
    '
    ' + Applications will need to set the cooperative level at this point.
    ' + Full-screen applications will probably need to set the display
    '   mode.
    ' + The primary surface should be created here.
```

```
    ' + The rendering surface must be created at this point. It is
    '   assumed in this code fragment that, once created, the rendering
    '   surface is referred to by the variable ddsRender.
    ' + If a depth-buffer is being used, it should be created here.
    ' + Direct3D device enumeration can be done at this point.

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

The preceding sample invokes the **Direct3D3.CreateDevice** method to create the Direct3D device. In the case of this sample, a Direct3D HAL device is created if the call is successful.

Note that the target DirectDraw rendering surface that your application creates must be created for use as a Direct3D rendering target. To do this, it must pass a **DDSURFACEDESC2** variable to the **DirectDraw4.CreateSurface** method. The **DDSURFACEDESC2** type has a member called **ddsCaps**, which is of type **DDSCAPS2**. The **DDSCAPS** type contains a member named **lCaps**, which must be set to DDSCAPS_3DDEVICE when **DirectDraw4.CreateSurface** is invoked.

The render target surface you use must be created in display memory (with the DDSCAPS_VIDEOMEMORY flag) when it will be use with a hardware-accelerated rendering device, and in system memory (using the DDSCAPS_SYSTEMMEMORY flag) otherwise.

Note

Some popular hardware devices require that the render target and depth buffer surfaces use the same bit depth. On such hardware, if your application uses a 16-bit render target surface, the attached depth buffer must also be 16-bits. For a 32-bit render target surface, the depth buffer must be 32-bits, of which 8-bits can be used for stencil buffering (if needed).
If the hardware upon which your application is running has this requirement, and your application fails to meet it, any attempts to create a rendering device that uses the non-compliant surfaces will fail.

Setting Transformations

[This is preliminary documentation and subject to change.]

[C++]
In C++, transformations are applied by using the **IDirect3DDevice3::SetTransform** method. For example, you could use code like this to set the view transformation:

```
HRESULT hr
D3DMATRIX  view;

// Fill in the view matrix.
```

```
hr = lpDev->SetTransform(D3DTRANSFORMSTATE_VIEW, &view);

if(FAILED(hr)){
   // Code to handle the error goes here.
}
```

There are three possible settings for the first parameter in a call to **IDirect3DDevice3::SetTransform**: D3DTRANSFORMSTATE_WORLD, D3DTRANSFORMSTATE_VIEW, and D3DTRANSFORMSTATE_PROJECTION. These transformation states are defined in the **D3DTRANSFORMSTATETYPE** enumerated type.

[Visual Basic]
In Visual Basic, transformations are applied by using the **Direct3DDevice3.SetTransform** method. For example, you could use code like this to set the view transformation:

```
On Local Error Resume Next
Dim view As D3DMATRIX

' Fill in the view matrix.
Call d3dDev.SetTransform(D3DTRANSFORMSTATE_VIEW, view)

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

There are three possible settings for the first parameter in a call to **Direct3DDevice3.SetTransform**: D3DTRANSFORMSTATE_WORLD, D3DTRANSFORMSTATE_VIEW, and D3DTRANSFORMSTATE_PROJECTION. These transformation states are defined in the **CONST_D3DTRANSFORMSTATETYPE** enumeration.

Rendering

[This is preliminary documentation and subject to change.]

Most Direct3D applications will use the DrawPrimitive family of methods to render a 3-D scene. The DrawPrimitive architecture is recommended over the legacy execute-buffer rendering model. As a result, the following topics present a DrawPrimitive-centered discussion about rendering:

- About Rendering
- Beginning and Ending a Scene
- Clearing Surfaces

- Rendering Primitives
- Primitive Types
- Render States

Although most applications will use the DrawPrimitive family of rendering methods, Direct3D still supports the use of execute buffers. For more information about rendering with execute buffers, see Execute Buffers. Note that execute buffers are not supported by DirectX for Visual Basic.

About Rendering

[This is preliminary documentation and subject to change.]

[C++]
Direct3D device interfaces later than the first device interface (**IDirect3DDevice**) support the DrawPrimitive rendering architecture. The DrawPrimitive family of methods—introduced in DirectX 5.0 with the **IDirect3DDevice2** interface, and extended by **IDirect3DDevice3**—can render different types of primitives such as points, lines, or collections of triangles. (Primitives are discussed in detail in Primitive Types.)

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

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

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

[C++,Visual Basic]

Beginning and Ending a Scene

[This is preliminary documentation and subject to change.]

[C++]
Applications written in C++ notify Direct3D that scene rendering is about to begin by calling the **IDirect3DDevice3::BeginScene** method. **BeginScene** causes the system to check its internal data structures, the availability and validity of rendering surfaces, and sets an internal flag to signal that a scene is in progress. After you begin a scene, you can call the various rendering methods to render the primitives or individual vertices that make up the objects in the scene. Attempts to call rendering methods when a scene is not in progress will fail, returning D3DERR_SCENE_NOT_IN_SCENE. For more information about the rendering methods, see Rendering Primitives.

After you complete the scene, call the **IDirect3DDevice3::EndScene** method. The **EndScene** method clears an internal flag that signals when a scene is in progress, flushes cached data, and verifies the integrity of the rendering surfaces.

All rendering methods must be bracketed by calls to the **BeginScene** and **EndScene** methods. If surfaces are lost, or if internal errors occur, the scene methods return error values. You must call **EndScene** for each time you call the **BeginScene** method, even if **BeginScene** fails. For example, some simple scene code might look like this:

```
HRESULT hr;

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

// Always close the scene, even if BeginScene fails.
hr = lpDevice->EndScene();
if(FAILED(hr))
    return hr;
```

You cannot embed scenes; that is, you must complete rendering a scene before you can begin another one. Calling **IDirect3DDevice3::EndScene** when **IDirect3DDevice3::BeginScene** has not been called will return the D3DERR_SCENE_NOT_IN_SCENE error value. Likewise, calling **BeginScene** when a previous scene has not been completed (with the **EndScene** method) results in the D3DERR_SCENE_IN_SCENE error.

You should not attempt to call GDI functions on DirectDraw surfaces, such as the render target or textures, while a scene is being rendered (between **BeginScene** and **EndScene** calls). Attempts to do so can prevent the results of the GDI operations from being visible. If your application uses GDI functions, make sure that all GDI calls are made outside of the scene functions.

[Visual Basic]
Visual Basic applications notify Direct3D that scene rendering is about to begin by calling the **Direct3DDevice3.BeginScene** method. **BeginScene** causes the system to check its internal data structures, the availability and validity of rendering surfaces, and sets an internal flag to signal that a scene is in progress. After you begin a scene, you can call the various rendering methods to render the primitives or individual vertices that make up the objects in the scene. Attempts to call rendering methods when a scene is not in progress will fail, raising the D3DERR_SCENE_NOT_IN_SCENE error. For more information about the rendering methods, see Rendering Primitives.

After you complete the scene, call the **Direct3DDevice3.EndScene** method. The **EndScene** method clears an internal flag that signals when a scene is in progress, flushes cached data, and verifies the integrity of the rendering surfaces.

All rendering methods must be bracketed by calls to the **BeginScene** and **EndScene** methods. If surfaces are lost, or if internal errors occur, the scene methods return error values. You must call **EndScene** for each time you call the **BeginScene** method, even if **BeginScene** fails. For example, some simple scene code might look like this:

```
On Local Error Resume Next

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

' Always close the scene, even if BeginScene fails.
Call d3dDev.EndScene
If Err.Number <> DD_OK Then
   ' Code to handle error goes here.
End If
```

You cannot embed scenes; that is, you must complete rendering a scene before you can begin another one. Calling **Direct3DDevice3.EndScene** when **Direct3DDevice3.BeginScene** has not been called will raise a D3DERR_SCENE_NOT_IN_SCENE error. Likewise, calling **BeginScene** when a previous scene has not been completed (with the **EndScene** method) results in the D3DERR_SCENE_IN_SCENE error.

You should not attempt to call GDI functions on DirectDraw surfaces, such as the render target or textures, while a scene is being rendered (between **BeginScene** and **EndScene** calls). Attempts to do so can prevent the results of the GDI operations from being visible. If your application uses GDI functions, make sure that all GDI calls are made outside of the scene functions.

Clearing Surfaces

[This is preliminary documentation and subject to change.]

Before rendering objects in a scene, you should clear the viewport on the render target surface (or a subset of the viewport). Clearing the viewport causes the system to set the desired portion of the render target surface and any attached depth or stencil buffers to a desired state. This resets the areas of the surface that will be rendered again and resets the corresponding areas of the depth and stencil buffers, if any are in use. Clearing a render target surface can set the desired region to a default color or texture. For depth and stencil buffers, this can set a depth or stencil value.

[C++]

The **IDirect3DViewport3** interface offers the **IDirect3DViewport3::Clear** and **IDirect3DViewport3::Clear2** methods to clear the viewport. For more information about using these methods, see Clearing a Viewport in the Viewports and Clipping section.

[Visual Basic]
The **Direct3DViewport3** Visual Basic class offers the **Direct3DViewport3.Clear** and **Direct3DViewport3.Clear2** methods to clear the viewport. For more information about using these methods, see Clearing a Viewport in the Viewports and Clipping section.

Optimization note:

Applications that render scenes covering the entire area of the render target surface can improve performance by clearing the attached depth and stencil buffer surfaces (if any) instead of the render target. In this case, clearing the depth buffer causes Direct3D to rewrite the render target on the next rendered frame, making an explicit clear operation on the render target redundant. However, if your application renders only to a portion of the render target surface, explicit clear operations are required.

Rendering Primitives

[This is preliminary documentation and subject to change.]

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

- The DrawPrimitive Methods
- Rendering Strided Vertices

The DrawPrimitive Methods

[This is preliminary documentation and subject to change.]

[C++]
Direct3D offers several methods to render primitives and indexed primitives in a single call or as individual vertices. All rendering methods in the **IDirect3DDevice3** interface accept a combination of flexible vertex format flags that describe the vertex type your application uses; these flags also determine which parts of the geometry pipeline the system will apply. (This differs from the version of the methods in the **IDirect3DDevice2**, which accept only the concrete vertex formats identified by members of the **D3DPRIMITIVETYPE** enumeration.) For more information about these descriptors, see Vertex Formats.

The DrawPrimitive family of rendering methods can be subdivided according to the style of primitive (non-indexed or indexed) a given method is capable of rendering. The system offers methods for both primitive styles, specified as groups of vertices or one vertex at a time. The following paragraphs reflect this subdivision:

Non-indexed and Indexed Primitive Methods

IDirect3DDevice3::DrawPrimitive, **IDirect3DDevice3::DrawPrimitiveStrided**, and **IDirect3DDevice3::DrawPrimitiveVB** render groups of non-indexed vertices, strided vertices, and non-indexed vertices contained within vertex buffers.

The **IDirect3DDevice3::DrawIndexedPrimitive**, **IDirect3DDevice3::DrawIndexedPrimitiveStrided**, and **IDirect3DDevice3::DrawIndexedPrimitiveVB** methods render vertex groups by indexing into a provided array of vertices, strided vertices, or by indexing into vertices within a vertex buffer.

Non-indexed and Indexed Vertex Methods

The **IDirect3DDevice3::Begin** method informs the system that you are beginning a sequence of vertices, and you can render the vertices with subsequent calls to **IDirect3DDevice3::Vertex**. When you are done with a sequence of vertices, you must call the **IDirect3DDevice3::End** method.

Similar to the non-indexed vertex rendering methods, the system offers the **IDirect3DDevice3::BeginIndexed** method to begin rendering vertices by their index, followed by calls to **IDirect3DDevice3::Index** for each vertex. (Just like non-indexed vertices, you finish the vertex sequence by calling **IDirect3DDevice3::End**.)

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

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

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

[Visual Basic]
Direct3D offers Visual Basic applications several methods to render primitives and indexed primitives in a single call or as individual vertices. All rendering methods in the **Direct3DDevice3** class accept a combination of flexible vertex format flags that describe the vertex type your application uses; these flags also determine which parts of the geometry pipeline the system will apply. For more information about these descriptors, see Vertex Formats.

The DrawPrimitive family of rendering methods can be subdivided according to the style of primitive (non-indexed or indexed) a given method is capable of rendering. The system offers methods for both primitive styles, specified as groups of vertices or one vertex at a time. The following paragraphs reflect this subdivision:

Non-indexed and Indexed Primitive Methods

Direct3DDevice3.DrawPrimitive and **Direct3DDevice3.DrawPrimitiveVB** render groups of non-indexed vertices alone or contained within vertex buffers.

The **Direct3DDevice3.DrawIndexedPrimitive** and **Direct3DDevice3.DrawIndexedPrimitiveVB** methods render vertex groups by indexing into a provided array of vertices or by indexing vertices within a vertex buffer.

Non-indexed and Indexed Vertex Methods

The **Direct3DDevice3.Begin** method informs the system that you are beginning a sequence of vertices, and you can render the vertices with subsequent calls to **Direct3DDevice3.Vertex**. When you are done with a sequence of vertices, you must call the **Direct3DDevice3.End** method.

Similar to the non-indexed vertex rendering methods, the system offers the **Direct3DDevice3.BeginIndexed** method to begin rendering vertices by their index, followed by calls to **Direct3DDevice3.Index** for each vertex. (Just like non-indexed vertices, you finish the vertex sequence by calling **IDirect3DDevice3::End**.)

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

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

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

Note:

DirectX for Visual Basic does not support rendering strided vertices.

Rendering Strided Vertices

[This is preliminary documentation and subject to change.]

[Visual Basic]

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

[C++]
Because of the indirection provided by strided vertices—detailed in the Strided Vertex Format section—applications must take care to set up the vertices properly. Often, developers forget to include a vertex component in the associated **D3DDRAWPRIMITIVESTRIDEDDATA** structures. This oversight doesn't necessarily cause the rendering methods to fail, but can result in "missing" geometry that is difficult to troubleshoot.

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

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

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

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

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

```
g_StridedData.diffuse.dwStride         = sizeof(MTVERTEX);
g_StridedData.textureCoords[0].dwStride = sizeof(MTVERTEX);
g_StridedData.textureCoords[1].dwStride = sizeof(MTVERTEX);

// Render the vertices with multiple texture blending (Modulate).
g_pd3dDevice->SetTextureStageState( 1, D3DTSS_COLOROP, D3DTOP_MODULATE );
g_pd3dDevice->SetTexture( 0, g_BaseTextureMap);
g_pd3dDevice->SetTexture( 1, g_LightMap);
g_pd3dDevice->DrawPrimitiveStrided( D3DPT_TRIANGLELIST,
                    D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX2,
                    &g_StridedData, 12, NULL );
```

Primitive Types

[This is preliminary documentation and subject to change.]

Direct3D can create and manipulate the following types of primitives:

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

[C++]
You render all of these primitive types from a C++ application with the **IDirect3DDevice3::DrawPrimitive** or **IDirect3DDevice3::DrawIndexedPrimitive** methods. For more information, see Rendering.

[Visual Basic]
A Visual Basic application renders these primitive types by calling the **Direct3DDevice3.DrawPrimitive** or **Direct3DDevice3.DrawIndexedPrimitive** methods. For more information, see Rendering.

Point Lists

[This is preliminary documentation and subject to change.]

A point list is a collection of vertices that are rendered as isolated points. Your application can use them in 3-D scenes for star fields, or dotted lines on the surface of a polygon. Applications create a point list by filling an array of vertices.

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

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

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

Render the point list using the **IDirect3DDevice3::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice3::DrawPrimitive** for drawing the point list in the preceding example. All calls to **IDirect3DDevice3::DrawPrimitive** must occur between **IDirect3DDevice3::BeginScene** and **IDirect3DDevice3::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice3 is a valid
// pointer to an IDirect3DDevice3 interface.
hResult =
   lpDirect3DDevice3->DrawPrimitive(D3DPT_POINTLIST,
                                    D3DFVF_VERTEX,
                                    lpVerts,
                                    TOTAL_VERTS,
                                    D3DDP_WAIT);
```

[Visual Basic]

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

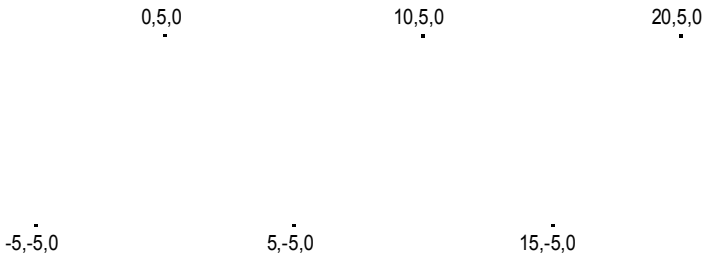
```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

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

Render the point list using the **IDirect3DDevice3::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice3::DrawPrimitive** for drawing the point list in the preceding example. All calls to **IDirect3DDevice3::DrawPrimitive** must occur between **IDirect3DDevice3::BeginScene** and **IDirect3DDevice3::EndScene**.

```
HRESULT hResult;
```

```
// This code fragment assumes that lpDirect3DDevice3 is a valid
// pointer to an IDirect3DDevice3 interface.
hResult =
   lpDirect3DDevice3->DrawPrimitive(D3DPT_POINTLIST,
                                    D3DFVF_VERTEX,
                                    lpVerts,
                                    TOTAL_VERTS,
                                    D3DDP_WAIT);
```

The following illustration shows the resulting points.



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

Line Lists

[This is preliminary documentation and subject to change.]

A line list is a list of isolated, straight line segments. Line lists are useful for such tasks as adding sleet or heavy rain to a 3-D scene.

Create a line list by filling an array of vertices, as in the following code fragment. The number of vertices in a line list must be greater than or equal to two, and it must be even.

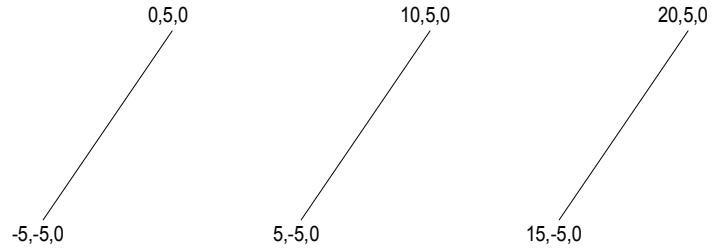
```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

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

Render the line list using the **IDirect3DDevice3::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice3::DrawPrimitive** for drawing the line list in the preceding example. Remember that all calls to

IDirect3DDevice3::DrawPrimitive must occur between **IDirect3DDevice3::BeginScene** and **IDirect3DDevice3::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice3 is a valid
// pointer to an IDirect3DDevice3 interface.
hResult =
   lpDirect3DDevice3 ->DrawPrimitive(D3DPT_LINELIST,
                                     D3DFVF_VERTEX,
                                     lpVerts,
                                     TOTAL_VERTS,
                                     D3DDP_WAIT);
```

The following illustration shows the resulting lines.



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

Line Strips

[This is preliminary documentation and subject to change.]

A line strip is a primitive that is composed of connected line segments. Your application can use line strips for creating polygons that are not closed. A closed polygon is a polygon whose last vertex is connected to its first vertex by a line segment. If your application makes polygons based on line strips, the vertices are not guaranteed to be coplanar.

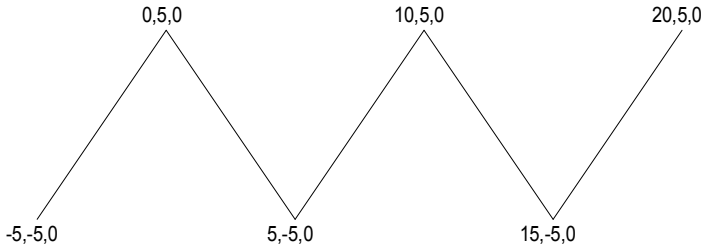
You create a line strip by filling an array of vertices, as in the following code fragment:

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

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

Render the line list using the **IDirect3DDevice3::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice3::DrawPrimitive** for drawing the line strip in the preceding example. Remember that all calls to **IDirect3DDevice3::DrawPrimitive** must occur between **IDirect3DDevice3::BeginScene** and **IDirect3DDevice3::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice3 is a valid
// pointer to an IDirect3DDevice3 interface.
hResult =
   lpDirect3DDevice3 ->DrawPrimitive(D3DPT_LINESTRIP,
                              D3DFVF_VERTEX,
                              lpVerts,
                              TOTAL_VERTS,
                              D3DDP_WAIT);
```

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



Triangle Lists

[This is preliminary documentation and subject to change.]

A triangle list is a list of isolated triangles. They may or may not be near each other. A triangle list must have at least three vertices. The total number of vertices must be divisible by three.

Use triangle lists when you want to create an object that is composed of disjoint pieces. For instance, one way to create a force field wall in a 3-D game is to specify a large list of small, unconnected triangles. Then apply a material or texture to the triangle list that emits light. Each triangle in the wall appears to glow. The scene behind the wall becomes partially visible through the gaps between the triangles, as players might expect when looking at a force field.

Triangle lists are also useful for creating primitives that have sharp edges and are shaded with Gouraud shading. See Face and Vertex Normal Vectors.

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

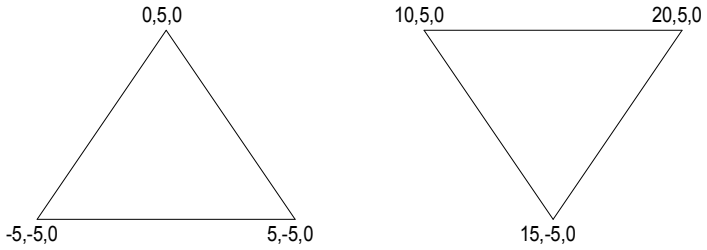
```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

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

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

Render a triangle list using the **IDirect3DDevice3::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice3::DrawPrimitive** for drawing the triangle list in the preceding example. Remember that all calls to **IDirect3DDevice3::DrawPrimitive** must occur between **IDirect3DDevice3::BeginScene** and **IDirect3DDevice3::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice3 is a valid
// pointer to an IDirect3DDevice3 interface.
hResult =
   lpDirect3DDevice3 ->DrawPrimitive(D3DPT_TRIANGLELIST,
                                     D3DFVF_VERTEX,
                                     lpVerts,
                                     TOTAL_VERTS,
                                     D3DDP_WAIT);
```

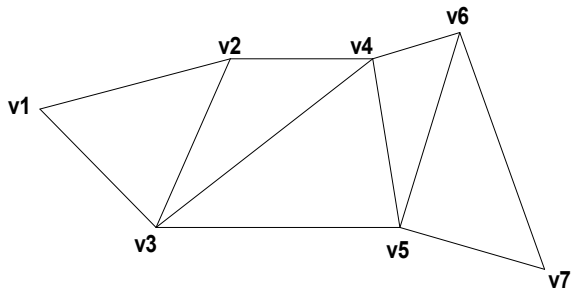
The following illustration depicts the resulting triangles.



Triangle Strips

[This is preliminary documentation and subject to change.]

A triangle strip is a series of connected triangles. Because the triangles are connected, the application does not need to repeatedly specify all three vertices for each triangle. For example, you only need seven vertices to define the following triangle strip.



The system uses vertices v1, v2, and v3 to draw the first triangle, v2, v4, and v3 to draw the second triangle, v3, v4, and v5 to draw the third, v4, v6, and v5 to draw the fourth, and so on. Notice that the vertices of the second and fourth triangles are out of order; this is required to make sure that all of the triangles are drawn in a clockwise orientation.

Most objects in 3-D scenes are composed of triangle strips. This is because triangle strips can be used to specify complex objects in a way that makes efficient use of memory and processing time. Triangle strips are also much easier to use than the **D3DTRIANGLE** structure that is used in conjunction with execute buffers.

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

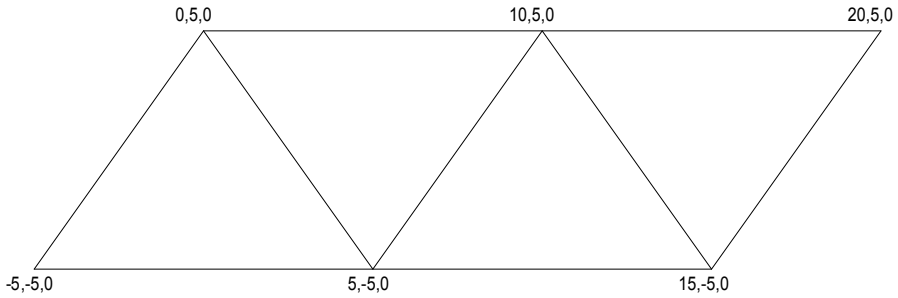
```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];

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

Your application can then render the triangle strip using the **IDirect3DDevice3::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice3::DrawPrimitive** for drawing the triangle strip in the preceding example. Remember that all calls to **IDirect3DDevice3::DrawPrimitive** must occur between **IDirect3DDevice3::BeginScene** and **IDirect3DDevice3::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice3 is a valid
// pointer to an IDirect3DDevice3 interface.
hResult =
   lpDirect3DDevice3 ->DrawPrimitive(D3DPT_TRIANGLESTRIP,
                                     D3DFVF_VERTEX,
                                     lpVerts,
```

```
                    TOTAL_VERTS,
                    D3DDP_WAIT);
```

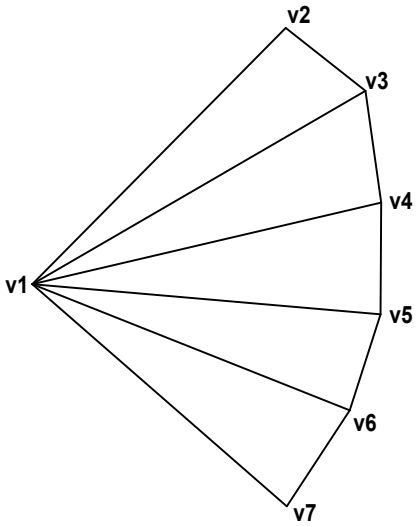
The following illustration shows the resulting triangle strip.



Triangle Fans

[This is preliminary documentation and subject to change.]

A triangle fan is similar to a triangle strip, except that all of the triangles share one vertex, as shown in the following illustration.



The system uses vertices v2, v3, and v1 to draw the first triangle, v3, v4, and v1 to draw the second triangle, v4, v5, and v1 to draw the third triangle, and so on. (When flat shading is enabled, the system shades the triangle with the color from its first vertex.)

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

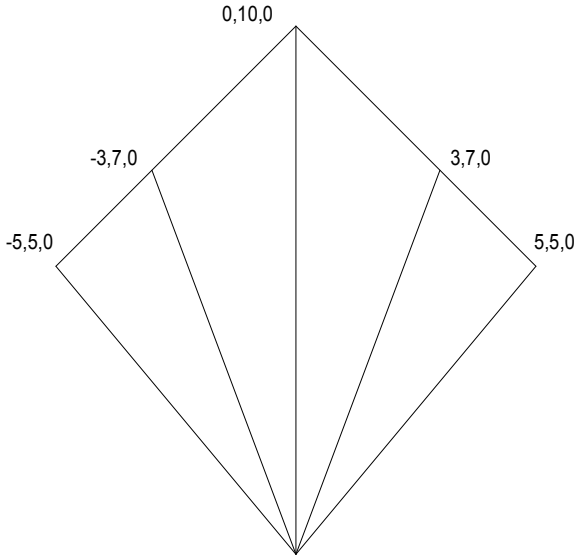
```
const DWORD TOTAL_VERTS=6;
D3DVERTEX lpVerts[TOTAL_VERTS];
```

```
lpVerts[0] = D3DVERTEX(D3DVECTOR(0,0,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[1] = D3DVERTEX(D3DVECTOR(-5,-5,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[2] = D3DVERTEX(D3DVECTOR(-3,7,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[3] = D3DVERTEX(D3DVECTOR(0,10,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[4] = D3DVERTEX(D3DVECTOR(3,7,0),D3DVECTOR(0,0,-1),0,0);
lpVerts[5] = D3DVERTEX(D3DVECTOR(5,5,0),D3DVECTOR(0,0,-1),0,0);
```

It can then render the triangle fan using the **IDirect3DDevice3::DrawPrimitive** method. The following code fragment illustrates the use of **IDirect3DDevice3::DrawPrimitive** for drawing the triangle fan in the preceding example. Remember that all calls to **IDirect3DDevice3::DrawPrimitive** must occur between **IDirect3DDevice3::BeginScene** and **IDirect3DDevice3::EndScene**.

```
HRESULT hResult;
// This code fragment assumes that lpDirect3DDevice3 is a valid
// pointer to an IDirect3DDevice3 interface.
hResult =
   lpDirect3DDevice3->DrawPrimitive(D3DPT_TRIANGLEFAN,
                                    D3DFVF_VERTEX,
                                    lpVerts,
                                    TOTAL_VERTS,
                                    D3DDP_WAIT);
```

This illustration depicts the resulting triangle fan.



Render States

[This is preliminary documentation and subject to change.]

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

- About Render States
- Current Texture
- Antialiasing States
- Texture Addressing State
- Texture Wrapping State
- Texture Borders
- Texture Perspective State
- Texture Filtering State
- Outline and Fill States
- Shading State
- Fog State
- Alpha States
- Texture Blending State
- Culling State
- Depth Buffering State
- Ramp State
- Subpixel Correction State
- Plane Masking State
- Color Keying State
- Render Command Batching State
- Stencil Buffer State

Note that the discussions in the topics listed here present techniques for controlling the rendering states when using the **IDirect3DDevice3** interface, which supports the DrawPrimitive methods. When using execute buffers under the **IDirect3DDevice** interface, the rendering states are controlled using the D3DOP_STATERENDER opcode.

About Render States

[This is preliminary documentation and subject to change.]

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

Applications control the other characteristics of the rendering state by invoking the **IDirect3DDevice3::SetRenderState** method. The **D3DRENDERSTATETYPE** enumerated type specifies all of the possible rendering states. Your application passes

a value from the **D3DRENDERSTATETYPE** enumeration as the first parameter to the **IDirect3DDevice3::SetRenderState** method.

Render states also can control the style of texturing and how texture filtering is done. For DirectX 6.0 and later, all texture-related render states are superseded by corresponding features offered by the **IDirect3DDevice3::SetTextureStageState** method. The superseded render states still work, but are mapped to affect the corresponding state in the first texture stage (stage 0). For best performance, applications should use the features in the **SetTextureStageState** method if favor of the legacy render states. The following list shows the superseded render states and lists the corresponding states offered in the new texturing model:

D3DRENDERSTATE_TEXTUREADDRESS
: Superseded by the D3DTSS_ADDRESS, D3DTSS_ADDRESSU, D3DTSS_ADDRESSV texture stage states for stage 0.

D3DRENDERSTATE_BORDERCOLOR
: Superseded by the D3DTSS_BORDERCOLOR texture stage state for stage 0.

D3DRENDERSTATE_TEXTUREMAG
: Superseded by the D3DTSS_MAGFILTER texture stage state for stage 0.

D3DRENDERSTATE_TEXTUREMIN
: Superseded by the D3DTSS_MINFILTER texture stage state for stage 0. Mipmap minification filtering is superseded by D3DTSS_MIPFILTER.

D3DRENDERSTATE_TEXTUREMAPBLEND
: Superseded by the D3DTSS_COLOROP and D3DTSS_ALPHAOP texture stage states for stage 0.

Current Texture

[This is preliminary documentation and subject to change.]

By default, Direct3D does not apply any textures to primitives being rendered. When your application selects a texture as the current texture, it instructs the Direct3D device to apply the texture to all primitives that are rendered from that time until the current texture is changed again. If each primitive in a 3-D scene has its own texture, the texture must be set before each primitive is rendered.

The **IDirect3D2** interface required the use of texture handles. With the **IDirect3D3** interface, textures are created as individual objects. An application accesses the functionality of textures through the **IDirect3DTexture2** interface. For information on obtaining a pointer to an **IDirect3DTexture2** interface, see Obtaining a Texture Interface Pointer. Your application can use a texture interface pointer to assign up to eight current textures. For more information, see Assigning the Current Textures.

If your application uses texture handles, it must pass the value D3DRENDERSTATE_TEXTUREHANDLE as the first parameter to **IDirect3DDevice3::SetRenderState**. The second parameter is the handle to the texture.

Applications can disable texturing by passing NULL as the second parameter to the **IDirect3DDevice3::SetRenderState** method.

Antialiasing States

[This is preliminary documentation and subject to change.]

Antialiasing is a method of making lines and edges appear smoother on the screen. Direct3D supports two ways to perform antialiasing, called edge antialiasing and full-scene antialiasing. For details about these techniques, see Antialiasing in the Common Techniques and Special Effects section.

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

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

Texture Addressing State

[This is preliminary documentation and subject to change.]

The U and V texture addressing states, set through the D3DRENDERSTATE_TEXTUREADDRESS, D3DRENDERSTATE_TEXTUREADDRESSU, and D3DRENDERSTATE_TEXTUREADDRESSV render states, have been superseded by the texture addressing features offered by the **IDirect3DDevice3** interface. However, applications that use the **IDirect3DDevice2** interface can still work with texture addressing modes as before.

Both approaches to texture addressing modes are discussed in Setting and Retrieving Texture Addressing Modes.

Texture Wrapping State

[This is preliminary documentation and subject to change.]

The U and V wrapping render states D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV have been superseded by the D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7 renderstates.

These new render states enable and disable U and V wrapping for various textures in the device's multitexture cascade. Set these render states to a combination of the D3DWRAP_U and D3DWRAP_V flags to enable wrapping in the corresponding direction, or omit use a value of zero to disable wrapping. Texture wrapping is disabled in all directions for all texture stages by default. For a conceptual overview, see Texture Wrapping.

Note

Although D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV are superseded, the **IDirect3DDevice3** interface still recognizes them. These older render states, when passed to the **IDirect3DDevice3::SetRenderState** affect U and V texture wrapping for the first texture stage (stage 0).

Texture Borders

[This is preliminary documentation and subject to change.]

The texture border color state has been superseded by the D3DTSS_BORDERCOLOR texture stage state supported by the **IDirect3DDevice3::SetTextureStageState** method. If your application uses **IDirect3DDevice3** interface, you should use **SetTextureStageState** to change the border color for each texture stage.

Applications that still use the legacy **IDirect3DDevice2** interface can work with texture border colors as before. In this case, set or retrieve the texture border color textures by passing the D3DRENDERSTATE_BORDERCOLOR enumerated value as the first parameter to **IDirect3DDevice2::SetRenderState**. The second parameter is the RGBA border color.

For more information, see About the Border Color Texture Address Mode.

Texture Perspective State

[This is preliminary documentation and subject to change.]

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

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

```
// This code fragment assumes that lpD3DDevice3 is a valid pointer to
// a Direct3DDevice3.

// Enable texture perspective.
lpD3DDevice3->SetRenderState(D3DRENDERSTATE_TEXTUREPERSPECTIVE,
                             TRUE);
```

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

Texture Filtering State

[This is preliminary documentation and subject to change.]

How your application sets texture filtering states depends largely on which version of the Direct3D device interface it uses. If your application uses the **IDirect3DDevice3** interface, the render states discussed here are effectively superseded by the texture filtering options offered by the **IDirect3DDevice3::SetTextureStageState** method. For a conceptual overview and more information on using **SetTextureStageState** for texture filtering, see Texture Filtering.

Note

Although the render states discussed here are superseded by texture stage states, the **IDirect3DDevice3::SetRenderState** (as opposed to the **IDirect3DDevice2** version) does not fail if you attempt to use them. Rather, the system maps the effects of these render states to the first stage in the multi-texture cascade, stage 0. Applications should not mix the legacy render states with their corresponding texture stage states, as unpredictable results can occur.

If your application is using the legacy **IDirect3DDevice2** interface, you set texture filtering states through the **IDirect3DDevice2::SetRenderState** method. Direct3D supports nearest point sampling, bilinear filtering, anisotropic texture filtering, and mipmap filtering. The filtering method is selected using the **D3DTEXTUREFILTER** enumerated type.

When an application magnifies a texture, it can use Direct3D devices to select a texture filtering method by passing the D3DRENDERSTATE_TEXTUREMAG enumerated value as the first parameter to the **IDirect3DDevice2::SetRenderState** method. It must pass one of the enumerated values in the **D3DTEXTUREFILTER** enumerated type as the second parameter. To select the filtering method used when the texture is being made smaller, set the first parameter of the **IDirect3DDevice2::SetRenderState** method to D3DRENDERSTATE_TEXTUREMIN. Set the second parameter to one of the enumerated values in the **D3DTEXTUREFILTER** enumerated type. When applications use software-emulated devices, they should use the same filtering methods for both D3DRENDERSTATE_TEXTUREMAG and D3DRENDERSTATE_TEXTUREMIN. Performance degradation will occur if they are not the same. Direct3D hardware devices (HAL and MMX) do not have this performance limitation.

Anisotropic filtering is disabled when the D3DRENDERSTATE_TEXTUREMAG state is set to D3DFILTER_NEAREST. Anisotropic filtering is only enabled when the D3DRENDERSTATE_TEXTUREMAG state is set to D3DFILTER_LINEAR. For the filter controlled by the D3DRENDERSTATE_TEXTUREMIN enumerated value, anisotropy is enabled when its state is set to D3DFILTER_LINEAR, D3DFILTER_MIPLINEAR, or D3DFILTER_LINEARMIPLINEAR.

Applications that use anisotropic texture filtering should set the degree of filtering to a value that is appropriate for their use. Anisotropic filtering is disabled when it is set to 1, and enabled by setting it to a value greater than 1. See Anisotropic Texture Filtering. and D3DRENDERSTATE_ANISOTROPY.

When using mipmap filtering, your application can select either mipmap near-point sampling or linear mipmap filtering. Mipmap near-point sampling selects the mipmap texture that most closely approximates the resolution of the final output texture, then uses nearest point sampling to obtain the color information. Linear mipmap filtering selects a color from the two closest mipmaps, then linearly interpolates color data between them. See Texture Filtering With Mipmaps and **D3DTEXTUREFILTER**.

Applications can perform special filtering effects by controlling the mipmap level of detail (LOD) bias. A positive bias on a mipmap texture results in a sharper, but more

aliased image. A negative bias causes a texture image to appear blurred. See D3DRENDERSTATE_MIPMAPLODBIAS.

Outline and Fill States

[This is preliminary documentation and subject to change.]

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

Direct3D uses the standard Windows ROP2 binary raster operations when it fills a primitive. The default value is R2_COPYPEN, which sets the pixel to the current pen color. For details, see **GetROP2** and **SetROP2** in the Platform SDK documentation. Although most applications will not need to change the default value, your application can change the raster fill operation by using the D3DRENDERSTATE_ROP2 enumerated value.

If you want your applications to enable dithering, it must pass the D3DRENDERSTATE_DITHERENABLE enumerated value as the first parameter to **IDirect3DDevice3::SetRenderState**. It must set the second parameter to TRUE to enable dithering, and FALSE to disable it.

A stippled fill pattern can be used if stippling is enabled. See D3DRENDERSTATE_STIPPLEENABLE. A 32x32 stipple pattern is specified using the enumerated values D3DRENDERSTATE_STIPPLEPATTERN00 through D3DRENDERSTATE_STIPPLEPATTERN31. Each of these enumerated values corresponds to one line of the stipple pattern. For example, to set the first line of the stipple pattern, pass D3DRENDERSTATE_STIPPLEPATTERN00 as the first parameter to **IDirect3DDevice3::SetRenderState**. Pass the hexadecimal value of the stipple pattern as the second parameter.

At times, drawing the last pixel in a line can cause unsightly overlap with surrounding primitives. This can be controlled using the D3DRENDERSTATE_LASTPIXEL enumerated value. However, this setting should not be altered without some forethought. Under some conditions, suppressing the rendering of the last pixel can cause unsightly gaps between primitives.

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

Shading State

[This is preliminary documentation and subject to change.]

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

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

```
// This code fragment assumes that lpD3DDevice3 is a valid pointer to
// a Direct3DDevice3.

// Set the shading state.
lpD3DDevice3->SetRenderState(D3DRENDERSTATE_SHADEMODE,
                             D3DSHADE_FLAT);
```

Fog State

[This is preliminary documentation and subject to change.]

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

The current Direct3D device enables and disables fog blending, controls fog color, and manipulates some fog parameters. You can enable fog by setting the D3DRENDERSTATE_FOGENABLE render state to TRUE. The fog color can be set to any **D3DCOLOR** value (the alpha component of the fog color is ignored). See D3DRENDERSTATE_FOGCOLOR.

For general information about using fog, see Fog, in the Common Techniques and Special Effects section.

Alpha States

[This is preliminary documentation and subject to change.]

The alpha value of a color controls its transparency. Enabling alpha blending allows colors, materials, and textures on a surface to be blended with transparency onto another surface. For more information, see Alpha Texture Blending and Multipass Texture Blending.

Alpha blending render states

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

The type of alpha blending that is done depends on the D3DRENDERSTATE_SRCBLEND and D3DRENDERSTATE_DESTBLEND render states. Source and destination blend states are used in pairs. The following code fragment demonstrates how the source blend state is set to D3DBLEND_SRCCOLOR and the destination blend state is set to D3DBLEND_INVSRCCOLOR.

```
// This code fragment assumes that lpD3DDevice3 is a valid pointer to
// an IDirect3DDevice3 interface.
```

```
// Set the source blend state.
lpD3DDevice3->SetRenderState(D3DRENDERSTATE_SRCBLEND,
                    D3DBLEND_SRCCOLOR);

// Set the destination blend state.
lpD3DDevice3->SetRenderState(D3DRENDERSTATE_DESTBLEND,
                    D3DBLEND_INVSRCCOLOR);
```

As a result of the calls in the preceding code fragment, Direct3D performs a linear blend between the source color (the color of the primitive being rendered at the current location) and the destination color (the color at the current location in the frame buffer). This gives an appearance similar to tinted glass. Some of the color of the destination object seems to be transmitted through the source object. The rest of it appears to be absorbed.

Altering the source and destination blend states can give the appearance of emissive objects in a foggy or dusty atmosphere. For instance, if your application models flames, force fields, plasma beams, or similarly radiant objects in a foggy environment, set the source and destination blend states to D3DBLEND_ONE.

Another application of alpha blending is controlling the lighting in a 3-D scene, also called light mapping. Setting the source blend state to D3DBLEND_ZERO and the destination blend state to D3DBLEND_SRCALPHA darkens a scene according to the source alpha information. The source primitive is used as a light map that scales the contents of the frame buffer to darken it when appropriate. This produces monochrome light mapping.

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

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

Alpha-testing render states

Applications can use alpha testing to control when pixels are written to the render target surface. By using the D3DRENDERSTATE_ALPHATESTENABLE enumerated value, your application sets the current Direct3D device so that it tests each pixel according to an alpha test function. If the test succeeds, the pixel is written to the surface. If it doesn't, Direct3D ignores it. Select the alpha test function with the D3DRENDERSTATE_ALPHAFUNC enumerated value. Your application can set a reference alpha value for all pixels to be compared against by using the D3DRENDERSTATE_ALPHAREF render state.

The most common use for alpha testing is to improve performance when rasterizing objects that are nearly transparent. If the color data being rasterized is more opaque than the color already at a given pixel (D3DPCMPCAPS_GREATEREQUAL) then the pixel is written, otherwise the rasterizer ignores the pixel altogether, saving the processing required to blend the two colors. The following example checks to see if a

given comparison function is supported and, if so, it sets the comparison function parameters required to improve performance during rendering.

```
// This example assumes that pd3dDeviceDesc is a
// D3DDEVICEDESC structure that was filled with a
// previous call to IDirect3DDevice3::GetCaps.
if (pd3dDeviceDesc->dpcTriCaps.dwAlphaCmpCaps & D3DPCMPCAPS_GREATEREQUAL)
{
   dev->SetRenderState( D3DRENDERSTATE_ALPHAREF, (DWORD)0x00000001);
   dev->SetRenderState( D3dRENDERSTATE_ALPHATESTENABLE, TRUE );
   dev->SetRenderState( D3DRS_ALPHACMP, D3DCMP_GREATEREQUAL );
}

// If the comparison isn't supported, render anyway.
// The only drawback is no performance gain.
```

Not all hardware supports all alpha testing features. You can check the device capabilities by calling the **IDirect3DDevice3::GetCaps** method. After retrieving the device capabilities, check the **dwAlphaCmpCaps** member of the **D3DPRIMCAPS** structure (contained by the associated **D3DDEVICEDESC** structure) for the desired comparison function. If the **dwAlphaCmpCaps** member contains only the D3DPCMPCAPS_ALWAYS capability or only the D3DPCMPCAPS_NEVER capability, the driver does not support alpha tests at all.

Texture Blending State

[This is preliminary documentation and subject to change.]

Your application can control the mode that is used to blend textures onto the surfaces of primitives. Applications that use texture handles set the texture blending state by invoking the **IDirect3DDevice3::SetRenderState** method and passing the enumerated value D3DRENDERSTATE_TEXTUREMAPBLEND as the first parameter. Pass a value from the **D3DTEXTUREBLEND** enumerated type as the second parameter.

Applications that use texture interface pointers set the texture blending state in the blending stages associated with the set of current textures. For more information, see Multiple Texture Blending.

Culling State

[This is preliminary documentation and subject to change.]

As Direct3D renders primitives, it culls those primitives that are facing away from the viewer. The culling mode of HAL and MMX devices can be set using the **D3DCULL** enumerated type. See D3DRENDERSTATE_CULLMODE. By default, Direct3D culls back faces with counterclockwise vertices.

The following code sample illustrates the processor setting the culling mode to cull back faces with clockwise vertices.

```
// This code fragment assumes that lpD3DDevice3 is a valid pointer to
```

```
// an IDirect3DDevice3 interface.

// Set the culling state.
lpD3DDevice3->SetRenderState(D3DRENDERSTATE_CULLMODE,
                             D3DCULL_CW);
```

Depth Buffering State

[This is preliminary documentation and subject to change.]

Depth buffering is a method of removing hidden lines and surfaces. For a conceptual overview, see What are Depth Buffers?. By default, Direct3D does not use depth buffering. You can update the depth buffering state with the D3DRENDERSTATE_ZENABLE render state, using one of the members of the **D3DZBUFFERTYPE** enumeration to specify the new state value.

If, for some reason, your application needs to prevent Direct3D from writing to the depth buffer, it can use the D3DRENDERSTATE_ZWRITEENABLE enumerated value, specifying FALSE as the second parameter for the call to **IDirect3DDevice3::SetRenderState**.

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

```
// This code fragment assumes that lpD3DDevice3 is a valid pointer to
// a Direct3DDevice3.

// Enable z-buffering.
lpD3DDevice3->SetRenderState(D3DRENDERSTATE_ZENABLE,
                             D3DZB_TRUE); // D3DZB_TRUE is the same as TRUE
```

Your application can also use the members of the **D3DCMPFUNC** enumerated type to select the comparison function that Direct3D uses when performing depth buffering. See D3DRENDERSTATE_ZFUNC.

Z-biasing is a method of displaying one surface in front of another even if their depth values are the same. You can use this technique for a variety of effects. A common example is rendering shadows on walls. Both the shadow and the wall have the same depth value. However, you want your application to show the shadow on the wall. Giving a z-bias to the shadow makes Direct3D display them properly (see D3DRENDERSTATE_ZBIAS).

Ramp State

[This is preliminary documentation and subject to change.]

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

Subpixel Correction State

[This is preliminary documentation and subject to change.]

The D3DRENDERSTATE_SUBPIXEL render state is not used, and the system ignores it.

Plane Masking State

[This is preliminary documentation and subject to change.]

Special effects can be achieved by using bit masks on the color channels. For instance, a scene with a strong, flashing red light can be simulated by selectively masking the blue and green channels so that they are darker, and the red channel so that it is brighter. Your program can turn the effect on and off alternately for intervals to simulate flashing.

To set the plane mask, call the **IDirect3DDevice3::SetRenderState** method with the first parameter set to D3DRENDERSTATE_PLANEMASK and the second set to the desired plane mask.

Note

The D3DRENDERSTATE_PLANEMASK render state is not supported by the software rasterizers, and is often ignored by hardware drivers. To disable writes to the color buffer by using alpha blending, set D3DRENDERSTATE_SRCBLEND to D3DBLEND_ZERO and D3DRENDERSTATE_DESTBLEND to D3DBLEND_ONE.

Color Keying State

[This is preliminary documentation and subject to change.]

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

Set color keys with the **IDirectDrawSurface4::SetColorKey** method for the surface that will be using the color key (in this case, the render target surface). However, color keying can be toggled on and off by calling the **IDirect3DDevice3::SetRenderState** method. Set the first parameter to D3DRENDERSTATE_COLORKEYENABLE. Your application should set the second parameter to TRUE to enable color keying and FALSE to disable it. The default is FALSE.

```
// This code fragment assumes that lpD3DDevice3 is a valid pointer to
// a Direct3DDevice3.

// Disable color keying.
lpD3DDevice3->SetRenderState(D3DRENDERSTATE_COLORKEYENABLE,
                             FALSE);
```

Render Command Batching State

[This is preliminary documentation and subject to change.]

By default, calls to **IDirect3DDevice2::DrawPrimitive** and **IDirect3DDevice2::DrawIndexedPrimitive** within a scene are batched. That is, they are buffered together and passed to the Direct3D device driver in one call. Changes in rendering states within the scene are also buffered. The buffer contents are passed to the device driver when the buffer is full or when the **IDirect3DDevice2::EndScene** method is invoked. This technique improves performance.

However, if an application makes change to the scene that are not changes to the rendering state, they may occur out of order with relation to the source code. For instance, if the contents of a texture are altered between calls to **IDirect3DDevice2::DrawPrimitive** or **IDirect3DDevice2::DrawIndexedPrimitive**, both primitives may be drawn with the new texture contents. Your application can resolve this problem by flushing the batching buffer before it makes scene changes that are not alterations of the rendering state.

An application flushes the batching buffer by calling **IDirect3DDevice2::SetRenderState** and passing the enumerated value D3DRENDERSTATE_FLUSHBATCH as the first parameter. The second parameter should be zero.

Note

This render state is only useful to applications that render with texture handles (using the **IDirect3DDevice2** interface). Batched primitives are implicitly flushed when rendering with the **IDirect3DDevice3** interface, as well as when rendering with execute buffers.

Stencil Buffer State

[This is preliminary documentation and subject to change.]

Applications use the stencil buffer to determine whether or not a pixel is written to the rendering target surface. For details, see Stencil Buffers.

Enable or disable stenciling by calling the **IDirect3DDevice3::SetRenderState** method. Pass D3DRENDERSTATE_STENCILENABLE as the value of the first parameter. Set the value of the second parameter to TRUE or FALSE to enable or disable it respectively.

Set the comparison function that Direct3D uses to perform the stencil test by invoking the **IDirect3DDevice3::SetRenderState** method. Set the value of the first parameter to D3DRENDERSTATE_STENCILFUNC. Pass a member of the **D3DCMPFUNC** enumerated type as the value of the second parameter.

The stencil reference value is the value in the stencil buffer that the stencil function uses for its test. By default, the stencil reference value is zero. Your application can set it by calling the **IDirect3DDevice3::SetRenderState** method. Pass D3DRENDERSTATE_STENCILREF as the value of the first parameter. Set the value of the second parameter to the new reference value.

Before Direct3D module performs the stencil test for any pixel, it does a bitwise AND of the stencil reference value and a stencil mask value. The result is then compared against the contents of the stencil buffer using the stencil comparison function. Your

application can set the stencil mask. Use the **IDirect3DDevice3::SetRenderState** method, and pass D3DRENDERSTATE_STENCILMASK as the value of the first parameter. Set the value of the second parameter to the new stencil mask.

To set the action that Direct3D takes when the stencil test fails, invoke the **IDirect3DDevice3::SetRenderState** method and pass D3DRENDERSTATE_STENCILFAIL as the first parameter. The second parameter must be a member of the **D3DSTENCILOP** enumerated type.

Your application can also control how Direct3D responds when the stencil test passes but the z-buffer test fails. Call the **IDirect3DDevice3::SetRenderState** method and pass D3DRENDERSTATE_STENCILZFAIL as the first parameter and use a member of the **D3DSTENCILOP** enumerated type for the second parameter.

In addition, your program can control what Direct3D does when both the stencil test and the z-buffer test pass. Use the **IDirect3DDevice3::SetRenderState** method and pass D3DRENDERSTATE_STENCILPASS as the first parameter. Again, the second parameter must be a member of the **D3DSTENCILOP** enumerated type.

Lighting States

[This is preliminary documentation and subject to change.]

In addition to the states discussed in Render States, Direct3D devices also control lighting states. When lighting states need to be changed, they are changed through the device interface. Therefore, your application can apply a particular rendering or lighting state to either a single primitive or a group of primitives.

The characteristics of the device lighting state are controlled by the **IDirect3DDevice3::SetLightState** method. All of the possible lighting states are specified by the **D3DLIGHTSTATETYPE** enumerated type, a member of which is passed as the first parameter to **IDirect3DDevice3::SetLightState**. The attributes of the lighting state are:

- Vertex Color Lighting
- Ambient Lighting
- Ramp Mode Lighting
- Vertex Fog Parameter States
- Material State

Vertex Color Lighting

[This is preliminary documentation and subject to change.]

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

With the introduction of the flexible vertex format, vertices may contain both vertex color and vertex normal information. By default, Direct3D uses this information when

it calculates lighting. If you want your application to disable the use of vertex color lighting information, invoke the **IDirect3DDevice3::SetLightState** method and pass D3DLIGHTSTATE_COLORVERTEX as the first parameter. Set the second parameter to FALSE to disable vertex color lighting, or TRUE to enable it.

If D3DLIGHTSTATE_COLORVERTEX is set to TRUE and a diffuse vertex color is present, the output alpha is equal to the diffuse alpha for the vertex. Otherwise, output alpha is equal to the alpha component of diffuse material, clamped to the range [0, 255].

Ambient Lighting

[This is preliminary documentation and subject to change.]

Ambient light is the surrounding light that comes from all directions. Your application sets the color of the ambient lighting by invoking the **IDirect3DDevice3::SetLightState** method and passing the enumerated value D3DLIGHTSTATE_AMBIENT as the first parameter. The second parameter is a color in RGBA format. The default is zero.

```
// This code fragment assumes that lpD3DDevice3 is a valid pointer to
// a Direct3DDevice3.

// Set the ambient light.
D3DCOLOR d3dclrAmbientLightColor = D3DRGBA(1.0f,1.0f,1.0f,1.0f);
lpD3DDevice3->SetLightState(D3DLIGHTSTATE_AMBIENT,
                            d3dclrAmbientLightColor);
```

Ambient light levels should not be confused with direct light. For more information, see Direct Light vs. Ambient Light.

Ramp Mode Lighting

[This is preliminary documentation and subject to change.]

The D3DLIGHTSTATE_COLORMODEL light state is not used. Your application implicitly chooses the color model it uses when it creates a rendering device.

Vertex Fog Parameter States

[This is preliminary documentation and subject to change.]

Because vertex fog is applied during lighting, you control several vertex fog parameters through the lighting interface. For general information about using vertex fog, see Vertex Fog, in the Common Techniques and Special Effects section.

The D3DLIGHTSTATE_FOGMODE, D3DLIGHTSTATE_FOGSTART, D3DLIGHTSTATE_FOGEND, and D3DLIGHTSTATE_FOGDENSITY lighting states control vertex fog parameters.

Material State

[This is preliminary documentation and subject to change.]

The D3DLIGHTSTATE_MATERIAL lighting state controls the current rendering material. By default, no material is selected. You can set a new rendering material by calling the **IDirect3DDevice3::SetLightState** method, passing D3DLIGHTSTATE_MATERIAL in the first parameter and the material handle for the material to be used as the second parameter. To deselect materials, set the second parameter to NULL. When no material is selected, the Direct3D lighting engine is disabled.

For more information, see Materials.

Emulation Modes

[This is preliminary documentation and subject to change.]

Direct3D furnishes three devices that emulate 3-D hardware in software, MMX, RGB, and ramp devices. For details, see RGB Device and Legacy Device Types. Because these devices perform software emulation, they render more slowly than 3-D hardware. However, if the user's computer has no 3-D hardware support, these software-emulated modes may be sufficient for your application.

MMX and RGB modes provides the full range of capabilities offered by the Direct3D transformation and lighting modules. In these modes, your application can use 8-, 16-, 24-, or 32-bit textures.

Ramp mode utilizes the full range of the transformation module but does not support colored lighting. A ramp mode device uses the Direct3D lighting module for monochrome gray-scale lighting only. Therefore, lights in ramp mode have intensity but no color. The gray-scale value is stored in the blue component of the light color.

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

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

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

AGP Surfaces and Direct3D Devices

[This is preliminary documentation and subject to change.]

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

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

Execute Buffers

[This is preliminary documentation and subject to change.]

This section provides an introduction to execute buffers. The following topics are discussed:

- About Execute Buffers
- Creating a Device for Execute Buffers
- Using Execute Buffers
- Triangle Flags
- Clip Tests on Execution
- Direct3D Execute-Buffer Tutorial

About Execute Buffers

[This is preliminary documentation and subject to change.]

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

Execute buffers are similar to the display lists you may be familiar with if you have experience with OpenGL programming. Execute buffers contain a vertex list followed by an instruction stream. The instruction stream consists of operation codes, or

opcodes, and the data that modifies those opcodes. Each execute buffer is bound to a single Direct3D device.

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

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

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

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

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

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

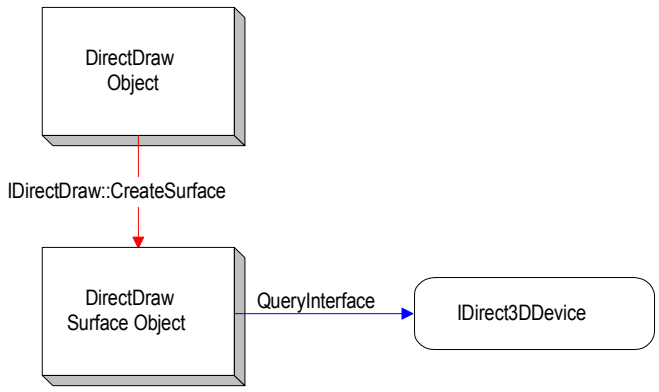
You can use the **IDirect3DExecuteBuffer** interface to get and set execute data, and to lock, unlock, optimize, and validate the execute buffer.

Creating a Device for Execute Buffers

[This is preliminary documentation and subject to change.]

To use execute buffer methods, your application must first initialize DirectDraw in the normal manner and obtain a pointer to the **IDirect3D3** interface. For details, see Accessing Direct3D. Your application should create a surface that includes the DDSCAPS_3DDEVICE capability. For information on creating surfaces, see Creating Surfaces. It must then call the **IUnknown::QueryInterface** method for that surface to obtain a pointer to an **IDirect3DDevice** interface. Use the **IDirect3DDevice::CreateExecuteBuffer** to create an execute buffer and obtain a

pointer to an **IDirect3DExecuteBuffer** interface. The following figure illustrates this process.



Using Execute Buffers

[This is preliminary documentation and subject to change.]

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

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

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

- Execute-Buffer Architecture
- Execute-Buffer Contents
- Creating an Execute Buffer
- Locking the Execute Buffer
- Filling the Execute Buffer
- Unlocking the Execute Buffer
- Executing the Execute Buffer
- States and State Overrides

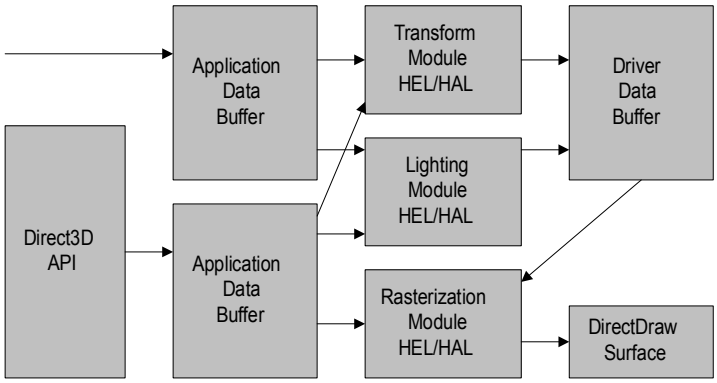
Execute-Buffer Architecture

[This is preliminary documentation and subject to change.]

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

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

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



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

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

Execute-Buffer Contents

[This is preliminary documentation and subject to change.]

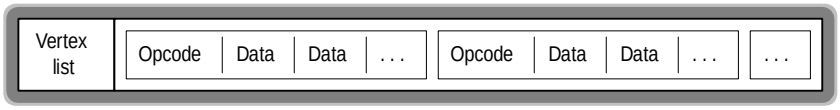
Execute buffers contain a list of vertices followed by stream of instructions about how to use those vertices. (All of these are **DWORD**-aligned.) This section contains information about working with vertices and instructions in an execute buffer. The following topics are discussed.

- Execute Buffer Format
- Execute Buffer Vertices
- Execute Buffer Instructions

Execute Buffer Format

[This is preliminary documentation and subject to change.]

The following illustration shows the format of execute buffers. Note that all data in an execute buffer must be **DWORD**-aligned.



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

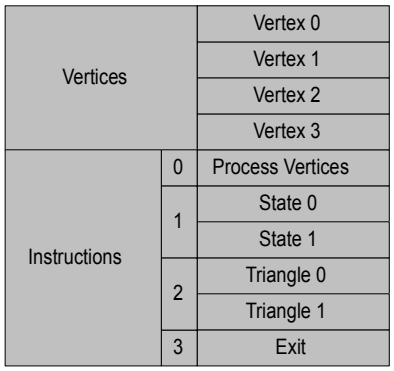
One of the most common instructions is a *triangle list* (D3DOP_TRIANGLE), which is simply a list of triangle primitives that reference vertices in the vertex list. Because all the primitives in the instruction stream reference vertices in the vertex list only, it is easy for the transformation module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

Execute Buffer Vertices

[This is preliminary documentation and subject to change.]

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

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



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

Execute Buffer Instructions

[This is preliminary documentation and subject to change.]

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

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

Execute buffer instructions are commands to the driver. Each instruction is identified by an operation code (opcode). All execute data is prefixed by an instruction header. Data accompanies each iteration of each opcode. Each opcode can have multiple arguments, including multiple triangles or multiple state changes. There are only a few main instruction types, as discussed in the following paragraphs:

Drawing Instructions

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

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

State-change Instructions

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

Transformation state	World, view and projection matrices
Light state	Surface material, fog, ambient lighting

Render state	Texture, antialiasing, z-buffering, and so on

Flow-control Instructions

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

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

Other Instructions

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

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

Creating an Execute Buffer

[This is preliminary documentation and subject to change.]

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

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

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

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

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

Locking the Execute Buffer

[This is preliminary documentation and subject to change.]

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

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

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

Filling the Execute Buffer

[This is preliminary documentation and subject to change.]

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

- Selecting the Vertex Type
- Triangles
- Processing Vertices
- Finishing the Instructions

You can streamline the task of filling execute buffers by taking advantage of the helper macros that ship with the samples in the DirectX SDK. The D3dmacs.h header file in the Misc directory of the samples contains many useful macros that will simplify your work. In particular, the macros **PUTD3DINSTRUCTION** and **VERTEX_DATA** are useful for filling execute buffers.

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

Selecting the Vertex Type

[This is preliminary documentation and subject to change.]

Applications may use all or part of the Direct3D rendering pipeline. The type of vertex that you use in your program determines how much of the rendering pipeline is used. For details, see Vertex Formats.

Triangles

[This is preliminary documentation and subject to change.]

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

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

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

Processing Vertices

[This is preliminary documentation and subject to change.]

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

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

Finishing the Instructions

[This is preliminary documentation and subject to change.]

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

Unlocking the Execute Buffer

[This is preliminary documentation and subject to change.]

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

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

Executing the Execute Buffer

[This is preliminary documentation and subject to change.]

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

The *dwFlags* parameter of **IDirect3DDevice::Execute** specifies whether the vertices you supply should be clipped. You should specify D3DEXECUTE_UNCLIPPED if all primitives in the buffer are contained within the viewport and D3DEXECUTE_CLIPPED otherwise.

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

States and State Overrides

[This is preliminary documentation and subject to change.]

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

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

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

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

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

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

D3DSHADE_GOURAUD is one of the members of the **D3DSHADEMODE** enumerated type.

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

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

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

Triangle Flags

[This is preliminary documentation and subject to change.]

When rendering with execute buffers, 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.

Clip Tests on Execution

[This is preliminary documentation and subject to change.]

Applications that use execute buffers can use the **IDirect3DDevice::Execute** method to render primitives with or without automatic clipping. Using this method without clipping is always faster than setting the clipping flags because clipping tests during either the transformation or rasterization stages slow the process. If your application does not use automatic clipping, however, it must ensure that all of the rendering data is wholly within the viewing frustum. The best way to ensure this is to use simple bounding volumes for the models and transform these first. You can use the results of this first transformation to decide whether to wholly reject the data because all the data is outside the frustum, whether to use the no-clipping version of the **IDirect3DDevice::Execute** method because all the data is within the frustum, or whether to use the clipping flags because the data is partially within the frustum. In Immediate Mode it is possible to set up this sort of functionality within one execute buffer by using the flags in the **D3DSTATUS** structure and the D3DOP_BRANCHFORWARD member of the **D3DOPCODE** enumerated type to skip geometry when a bounding volume is outside the frustum. Direct3D Retained Mode automatically uses these features to speed up its use of execute buffers.

Direct3D Execute-Buffer Tutorial

[This is preliminary documentation and subject to change.]

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

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

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

- Main Window Procedure
- WinMain Function

Definitions, Prototypes, and Globals

[This is preliminary documentation and subject to change.]

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

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

Header and Includes

[This is preliminary documentation and subject to change.]

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

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

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

#include "resource.h"
```

Constants in Imsample.c

[This is preliminary documentation and subject to change.]

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

#define WINDOW_CLASSNAME      "D3DSample1Class"

// Title for the application's window.

#define WINDOW_TITLE          "D3D Sample 1"

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

#define PAUSED_STRING         "Paused"

// Half height of the view window.

#define HALF_HEIGHT           D3DVAL(0.5)

// Front and back clipping planes.

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

// Fixed window size.

#define WINDOW_WIDTH          320
#define WINDOW_HEIGHT         200

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

#define MAX_DEVICE_NAME       256
#define MAX_DEVICE_DESC       256

// Amount to rotate per frame.

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

// Execute buffer contents

#define NUM_VERTICES          3
#define NUM_INSTRUCTIONS      6
```

```
#define NUM_STATES          7
#define NUM_PROCESSVERTICES   1
#define NUM_TRIANGLES       1
```

Macros in Imsample.c

[This is preliminary documentation and subject to change.]

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

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

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

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

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

Global Variables

[This is preliminary documentation and subject to change.]

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

static HINSTANCE            hAppInstance            = NULL;

// Running in debug mode?

static BOOL                 fDebug                  = FALSE;

// Is the application active?

static BOOL                 fActive                 = TRUE;

// Has the application been suspended?

static BOOL                 fSuspended              = FALSE;

// DirectDraw interfaces

static LPDIRECTDRAW         lpdd                    = NULL;
static LPDIRECTDRAWSURFACE  lpddPrimary             = NULL;
static LPDIRECTDRAWSURFACE  lpddDevice              = NULL;
```

```
static LPDIRECTDRAWSURFACE     lpddZBuffer             = NULL;
static LPDIRECTDRAWPALETTE     lpddPalette             = NULL;

// Direct3D interfaces

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

// Direct3D handles

static D3DMATRIXHANDLE         hd3dWorldMatrix         = 0;
static D3DMATRIXHANDLE         hd3dViewMatrix          = 0;
static D3DMATRIXHANDLE         hd3dProjMatrix          = 0;
static D3DMATERIALHANDLE       hd3dSurfaceMaterial     = 0;
static D3DMATERIALHANDLE       hd3dBackgroundMaterial  = 0;

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

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

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

static RECT                    rDstRect;

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

static RECT                    rSrcRect;
```

```
// Angle of rotation of the world matrix.

static double               dAngleOfRotation        = 0.0;

// Predefined transformations.

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

static D3DMATRIX d3dViewMatrix =
{
   D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 0.0),
   D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0),
   D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0),
   D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 5.0), D3DVAL( 1.0)
};

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

Function Prototypes

[This is preliminary documentation and subject to change.]

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

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

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

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

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

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

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

static LRESULT      RestoreSurfaces(void);

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

static HRESULT      UpdateViewport(void);

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

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

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

Enumerating Direct3D Devices

[This is preliminary documentation and subject to change.]

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

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

- Enumeration Callback Function
- Enumeration Function

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

Enumeration Callback Function

[This is preliminary documentation and subject to change.]

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

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

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

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

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

```
static HRESULT WINAPI
EnumDeviceCallback(LPGUID        lpGUID,
                   LPSTR         lpszDeviceDesc,
                   LPSTR         lpszDeviceName,
                   LPD3DDEVICEDESC lpd3dHWDeviceDesc,
                   LPD3DDEVICEDESC lpd3dSWDeviceDesc,
```

```
                LPVOID          lpUserArg)
{
    BOOL            fIsHardware;
    LPD3DDEVICEDESC lpd3dDeviceDesc;

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

    USE_PARAM(lpUserArg);

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

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

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

    if (fDebug && fIsHardware)
        return D3DENUMRET_OK;

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

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

        return D3DENUMRET_OK;
    }

        // The device must support Gouraud-shaded triangles.

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

            return D3DENUMRET_OK;
        }
    }
    else
    {
```

```
    if (!(lpd3dDeviceDesc->dpcTriCaps.dwShadeCaps &
         D3DPSHADECAPS_COLORGOURAUDRGB))
    {
                // No Gouraud shading. Skip this device.

        return D3DENUMRET_OK;
    }
}

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

    return D3DENUMRET_OK;
}

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

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

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

if (fIsHardware)
    return D3DENUMRET_CANCEL;

    // Otherwise, keep looking.

return D3DENUMRET_OK;
}
```

Enumeration Function

[This is preliminary documentation and subject to change.]

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

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

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

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

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

    dwDeviceBitDepth =
        BitDepthToFlags(ddsd.ddpfPixelFormat.dwRGBBitCount);

        // Enumerate the devices and pick one.

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

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

        return DDERR_NOTFOUND;
    }

    return DD_OK;
}
```

Creating Objects and Interfaces

[This is preliminary documentation and subject to change.]

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

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

Creating the Primary Surface and Clipper Object

[This is preliminary documentation and subject to change.]

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

```
static HRESULT
CreatePrimary(HWND hwnd)
{
    HRESULT            hRes;
    DDSURFACEDESC      ddsd;
    LPDIRECTDRAWCLIPPER lpddClipper;
    HDC                hdc;
    int                i;
    PALETTEENTRY       peColorTable[256];

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

    // Create the primary surface.

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

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

    hRes = DirectDrawCreateClipper(0, &lpddClipper, NULL);
    if (FAILED(hRes))
        return hRes;
```

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

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

lpddClipper->lpVtbl->Release(lpddClipper);

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

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

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

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

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

        for (i = 0; i < 10; i++)
            peColorTable[i].peFlags = D3DPAL_READONLY;
        for (i = 10; i < 246; i++)
            peColorTable[i].peFlags = D3DPAL_FREE | PC_RESERVED;
        for (i = 246; i < 256; i++)
            peColorTable[i].peFlags = D3DPAL_READONLY;
        hRes = lpdd->lpVtbl->CreatePalette(lpdd,
            DDPCAPS_8BIT, peColorTable, &lpddPalette, NULL);

        if (FAILED(hRes))
            return hRes;

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

    return DD_OK;
}
```

Creating the Direct3D Object

[This is preliminary documentation and subject to change.]

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

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

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

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

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

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

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

        // Retrieve the Direct3D interface to the DirectDraw/3D driver
        // object.

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

    return DD_OK;
}
```

Creating the Direct3D Device

[This is preliminary documentation and subject to change.]

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

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

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

    ASSERT(NULL != lpdd);
```

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

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

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

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

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

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

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

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

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

```
if (FAILED(hRes))
    return hRes;

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

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

// We now determine whether or not we need a z-buffer and, if
// so, its bit depth.

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

    dwZBufferBitDepth =
        FlagsToBitDepth(lpd3dDeviceDesc->dwDeviceZBufferBitDepth);

    // Create the z-buffer.

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

        // Attach it to the rendering target.
```

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

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

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

  return DD_OK;
}
```

Creating the Scene

[This is preliminary documentation and subject to change.]

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

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

    ASSERT(NULL != lpd3d);
    ASSERT(NULL != lpd3dDevice);
    ASSERT(NULL == lpd3dViewport);
    ASSERT(NULL == lpd3dMaterial);
    ASSERT(NULL == lpd3dBackgroundMaterial);
    ASSERT(NULL == lpd3dExecuteBuffer);
```

```
ASSERT(NULL == lpd3dLight);
ASSERT(0  == hd3dWorldMatrix);
ASSERT(0  == hd3dViewMatrix);
ASSERT(0  == hd3dProjMatrix);

// Create the light.

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

ZeroMemory(&d3dLight, sizeof(d3dLight));
d3dLight.dwSize = sizeof(d3dLight);
d3dLight.dltType = D3DLIGHT_POINT;
d3dLight.dcvColor.dvR    = D3DVAL( 1.0);
d3dLight.dcvColor.dvG    = D3DVAL( 1.0);
d3dLight.dcvColor.dvB    = D3DVAL( 1.0);
d3dLight.dcvColor.dvA    = D3DVAL( 1.0);
d3dLight.dvPosition.dvX  = D3DVAL( 1.0);
d3dLight.dvPosition.dvY  = D3DVAL(-1.0);
d3dLight.dvPosition.dvZ  = D3DVAL(-1.0);
d3dLight.dvAttenuation0  = D3DVAL( 1.0);
d3dLight.dvAttenuation1  = D3DVAL( 0.1);
d3dLight.dvAttenuation2  = D3DVAL( 0.0);
hRes = lpd3dLight->lpVtbl->SetLight(lpd3dLight, &d3dLight);
if (FAILED(hRes))
   return hRes;

// Create the background material.

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

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

```
d3dMaterial.dvPower      = D3DVAL(0.0);

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

d3dMaterial.dwRampSize    = 1;

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

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

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

// Create the matrices.

hRes = lpd3dDevice->lpVtbl->CreateMatrix(lpd3dDevice,
   &hd3dWorldMatrix);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dDevice->lpVtbl->SetMatrix(lpd3dDevice, hd3dWorldMatrix,
   &d3dWorldMatrix);
if (FAILED(hRes))
   return hRes;
hRes = lpd3dDevice->lpVtbl->CreateMatrix(lpd3dDevice,
   &hd3dViewMatrix);
```

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

// Create the surface material.

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

// Base green with white specular.

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

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

hRes = lpd3dMaterial->lpVtbl->GetHandle(lpd3dMaterial, lpd3dDevice,
   &hd3dSurfaceMaterial);
```

```
    if (FAILED(hRes))
        return hRes;

    // Build the execute buffer.

    dwVertexSize      = (NUM_VERTICES       * sizeof(D3DVERTEX));
    dwInstructionSize = (NUM_INSTRUCTIONS   * sizeof(D3DINSTRUCTION)) +
                        (NUM_STATES         * sizeof(D3DSTATE))     +
                        (NUM_PROCESSVERTICES *
                                sizeof(D3DPROCESSVERTICES))   +
                        (NUM_TRIANGLES      * sizeof(D3DTRIANGLE));
    dwExecuteBufferSize = dwVertexSize + dwInstructionSize;
    ZeroMemory(&d3dExecuteBufferDesc, sizeof(d3dExecuteBufferDesc));
    d3dExecuteBufferDesc.dwSize       = sizeof(d3dExecuteBufferDesc);
    d3dExecuteBufferDesc.dwFlags      = D3DDEB_BUFSIZE;
    d3dExecuteBufferDesc.dwBufferSize = dwExecuteBufferSize;
    hRes = lpd3dDevice->lpVtbl->CreateExecuteBuffer(lpd3dDevice,
        &d3dExecuteBufferDesc, &lpd3dExecuteBuffer, NULL);
    if (FAILED(hRes))
        return hRes;

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

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

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

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

    return DD_OK;
}
```

Filling the Execute Buffer

[This is preliminary documentation and subject to change.]

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

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

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

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

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

    // Lock the execute buffer.

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

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

```
lpVertex = (LPD3DVERTEX)d3dExeBufDesc.lpData;

// First vertex.

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

// Second vertex.

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

// Third vertex.

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

// Transform state - world, view and projection.

lpInstruction = (LPD3DINSTRUCTION)lpVertex;
lpInstruction->bOpcode = D3DOP_STATETRANSFORM;
lpInstruction->bSize   = sizeof(D3DSTATE);
lpInstruction->wCount  = 3U;
lpInstruction++;
lpState = (LPD3DSTATE)lpInstruction;
```

```
lpState->dtstTransformStateType = D3DTRANSFORMSTATE_WORLD;
lpState->dwArg[0] = hd3dWorldMatrix;
lpState++;
lpState->dtstTransformStateType = D3DTRANSFORMSTATE_VIEW;
lpState->dwArg[0] = hd3dViewMatrix;
lpState++;
lpState->dtstTransformStateType = D3DTRANSFORMSTATE_PROJECTION;
lpState->dwArg[0] = hd3dProjMatrix;
lpState++;

// Lighting state.

lpInstruction = (LPD3DINSTRUCTION)lpState;
lpInstruction->bOpcode = D3DOP_STATELIGHT;
lpInstruction->bSize   = sizeof(D3DSTATE);
lpInstruction->wCount  = 2U;
lpInstruction++;
lpState = (LPD3DSTATE)lpInstruction;
lpState->dlstLightStateType = D3DLIGHTSTATE_MATERIAL;
lpState->dwArg[0] = hd3dSurfaceMaterial;
lpState++;
lpState->dlstLightStateType = D3DLIGHTSTATE_AMBIENT;
lpState->dwArg[0] = RGBA_MAKE(128, 128, 128, 128);
lpState++;

// Render state.

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

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

```
    // the instruction is D3DPROCESSVERTICES_TRANSFORMLIGHT.

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

    // Draw the triangle.

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

    // Stop execution of the buffer.

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

    // Unlock the execute buffer.

    lpd3dExecuteBuffer->lpVtbl->Unlock(lpd3dExecuteBuffer);

    return DD_OK;
}
```

Animating the Scene

[This is preliminary documentation and subject to change.]

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

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

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

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

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

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

    return DD_OK;
}
```

Rendering Using an Execute Buffer

[This is preliminary documentation and subject to change.]

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

- Rendering the Scene
- Rendering a Single Frame

Rendering the Scene

[This is preliminary documentation and subject to change.]

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

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

For maximum efficiency we only want to clear those regions of the device surface and z-buffer which we actually rendered to in the last frame. This is the purpose of

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

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

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

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

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

    // Clear both back and z-buffer.

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

    // Start the scene.

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

    // Submit the execute buffer.

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

```
    // D3DEXECUTE_CLIPPED.

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

        // End the scene.

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

    return DD_OK;
}
```

Rendering a Single Frame

[This is preliminary documentation and subject to change.]

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

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

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

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

    while (TRUE)
    {
        hRes = RenderScene();
        if (SUCCEEDED(hRes))
        {
            hRes = lpddPrimary->lpVtbl->Blt(lpddPrimary, &rDstRect,
                lpddDevice, &rSrcRect, DDBLT_WAIT, NULL);
```

```
        if (SUCCEEDED(hRes))  // If it worked.
            return hRes;
    }
    while (DDERR_SURFACELOST == hRes)  // Restore lost surfaces
        hRes = RestoreSurfaces();
    if (FAILED(hRes))  // handle other failure cases
        return hRes;
  }
}
```

Working with Matrices

[This is preliminary documentation and subject to change.]

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

- Setting the Perspective Transformation
- Setting a Rotation Transformation

Setting the Perspective Transformation

[This is preliminary documentation and subject to change.]

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

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

    ASSERT(NULL != lpd3dMatrix);

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

    lpd3dMatrix->_11 =  D3DVAL(2.0);
    lpd3dMatrix->_12 =  D3DVAL(0.0);
    lpd3dMatrix->_13 =  D3DVAL(0.0);
    lpd3dMatrix->_14 =  D3DVAL(0.0);
    lpd3dMatrix->_21 =  D3DVAL(0.0);
```

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

Setting a Rotation Transformation

[This is preliminary documentation and subject to change.]

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

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

    ASSERT(NULL != lpd3dMatrix);

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

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

```
}
```

Restoring and Redrawing

[This is preliminary documentation and subject to change.]

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

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

Restoring the Direct3D Device

[This is preliminary documentation and subject to change.]

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

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

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

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

    return DD_OK;
}
```

Restoring the Primary Surface

[This is preliminary documentation and subject to change.]

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

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

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

Restoring All Surfaces

[This is preliminary documentation and subject to change.]

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

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

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

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

    return DD_OK;
}
```

Redrawing on Window Movement

[This is preliminary documentation and subject to change.]

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

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

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

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

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

        // Repaint the client area.

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

    return 0L;
}
```

Redrawing on Window Resizing

[This is preliminary documentation and subject to change.]

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

        // Nothing to do if we are suspended.

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

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

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

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

    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
    hRes = lpddDevice->lpVtbl->GetSurfaceDesc(lpddDevice,
        &ddsd);
    if (FAILED(hRes))
    {
        FatalError(hwnd, IDS_ERRMSG_DEVICESIZE, hRes);
        return 0L;
    }

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

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

        ReleaseScene();
        ReleaseDevice();
    }
}

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

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

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

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

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

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

    return 0L;
}
```

Repainting the Client Area

[This is preliminary documentation and subject to change.]

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

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

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

    USE_PARAM(lpps);
```

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

        PaintSuspended(hwnd, hdc);
    }

    return 0L;
}
```

Updating the Viewport

[This is preliminary documentation and subject to change.]

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

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

    ASSERT(NULL != lpd3dViewport);

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

Releasing Objects

[This is preliminary documentation and subject to change.]

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

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

Releasing the Direct3D Object

[This is preliminary documentation and subject to change.]

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

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

    return DD_OK;
}
```

Releasing the Direct3D Device

[This is preliminary documentation and subject to change.]

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

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

```
    {
        lpddZBuffer->lpVtbl->Release(lpddZBuffer);
        lpddZBuffer = NULL;
    }
    if (NULL != lpddDevice)
    {
        lpddDevice->lpVtbl->Release(lpddDevice);
        lpddDevice = NULL;
    }

    return DD_OK;
}
```

Releasing the Primary Surface

[This is preliminary documentation and subject to change.]

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

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

    return DD_OK;
}
```

Releasing the Objects in the Scene

[This is preliminary documentation and subject to change.]

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

```
static HRESULT
ReleaseScene(void)
{
    if (NULL != lpd3dExecuteBuffer)
    {
        lpd3dExecuteBuffer->lpVtbl->Release(lpd3dExecuteBuffer);
        lpd3dExecuteBuffer = NULL;
```

```
    }
    if (NULL != lpd3dBackgroundMaterial)
    {
        lpd3dBackgroundMaterial->
            lpVtbl->Release(lpd3dBackgroundMaterial);
        lpd3dBackgroundMaterial = NULL;
    }
    if (NULL != lpd3dMaterial)
    {
        lpd3dMaterial->lpVtbl->Release(lpd3dMaterial);
        lpd3dMaterial = NULL;
    }
    if (0 != hd3dWorldMatrix)
    {
        lpd3dDevice->lpVtbl->DeleteMatrix(lpd3dDevice, hd3dWorldMatrix);
        hd3dWorldMatrix = 0;
    }
    if (0 != hd3dViewMatrix)
    {
        lpd3dDevice->lpVtbl->DeleteMatrix(lpd3dDevice, hd3dViewMatrix);
        hd3dViewMatrix = 0;
    }
    if (0 != hd3dProjMatrix)
    {
        lpd3dDevice->lpVtbl->DeleteMatrix(lpd3dDevice, hd3dProjMatrix);
        hd3dProjMatrix = 0;
    }
    if (NULL != lpd3dLight)
    {
        lpd3dLight->lpVtbl->Release(lpd3dLight);
        lpd3dLight = NULL;
    }
    if (NULL != lpd3dViewport)
    {
        lpd3dViewport->lpVtbl->Release(lpd3dViewport);
        lpd3dViewport = NULL;
    }

    return DD_OK;
}
```

Error Checking

[This is preliminary documentation and subject to change.]

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

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

Checking for Active Status

[This is preliminary documentation and subject to change.]

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

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

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

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

   return 0L;
}
```

Reporting Standard Errors

[This is preliminary documentation and subject to change.]

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

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

```
    char szError[128];
    int  nStrID;

    // Turn the animation loop off.

    fSuspended = TRUE;

    // Get the high level error message.

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

    // We issue sensible error messages for common run time errors. For
    // errors which are internal or coding errors we simply issue an
    // error number (they should never occur).

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

    // Show the "paused" display.

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

    // Convert the error code into a string.

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

Reporting Fatal Errors

[This is preliminary documentation and subject to change.]

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

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

   DestroyWindow(hwnd);
}
```

Displaying a Notification String

[This is preliminary documentation and subject to change.]

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

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

   // Black background.

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

   // White text.

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

   GetClientRect(hwnd, &rect);

   // Clear the client area.
```

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

    // Draw the string centered in the client area.

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

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

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

Converting Bit Depths

[This is preliminary documentation and subject to change.]

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

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

Converting a Bit Depth into a Flag

[This is preliminary documentation and subject to change.]

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

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

```
}
```

Converting a Flag into a Bit Depth

[This is preliminary documentation and subject to change.]

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

```
static DWORD
FlagsToBitDepth(DWORD dwFlags)
{
    if (dwFlags & DDBD_1)
        return 1;
    else if (dwFlags & DDBD_2)
        return 2;
    else if (dwFlags & DDBD_4)
        return 4;
    else if (dwFlags & DDBD_8)
        return 8;
    else if (dwFlags & DDBD_16)
        return 16;
    else if (dwFlags & DDBD_24)
        return 24;
    else if (dwFlags & DDBD_32)
        return 32;
    else
        return 0;
}
```

Main Window Procedure

[This is preliminary documentation and subject to change.]

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

    switch (msg)
    {
        case WM_CREATE:
            hRes = CreateDirect3D(hwnd);
```

```
    if (FAILED(hRes))
    {
        ReportError(hwnd, IDS_ERRMSG_CREATEDEVICE, hRes);
        ReleaseDirect3D();
        return -1L;
    }

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

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

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

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

    return 0L;

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

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

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

    return 1L;
```

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

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

    EndPaint(hwnd, &ps);
    return lResult;

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

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

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

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

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

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

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

```
            }
            break;

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

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

            ReleaseScene();
            ReleaseDevice();
            ReleasePrimary();
            ReleaseDirect3D();

            PostQuitMessage(0);
            return 0L;
    }

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

WinMain Function

[This is preliminary documentation and subject to change.]

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

    USE_PARAM(hPrevInstance);

    // Record the instance handle.

    hAppInstance = hInstance;

    // Very simple command-line processing. We only have one
    // option - debug - so we will just assume that if anything was
    // specified on the command line the user wants debug mode.
```

```
// (In debug mode there is no hardware and all surfaces are
// explicitly in system memory.)

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

// Register the window class.

wndClass.style         = 0;
wndClass.lpfnWndProc   = WndProc;
wndClass.cbClsExtra    = 0;
wndClass.cbWndExtra    = 0;
wndClass.hInstance     = hInstance;
wndClass.hIcon         = LoadIcon(hAppInstance,
   MAKEINTRESOURCE(IDI_APPICON));
wndClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
wndClass.hbrBackground = GetStockObject(WHITE_BRUSH);
wndClass.lpszMenuName  = NULL;
wndClass.lpszClassName = WINDOW_CLASSNAME;

RegisterClass(&wndClass);

// Create the main window of the instance.

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

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

// The main message dispatch loop.

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

while (TRUE)
{
```

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

            if (WM_QUIT == msg.message)
            {
                break;
            }
            else
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        }
        else
        {
            // Animate the scene.

            OnIdle(hwnd);
        }
    }

    return msg.wParam;
}
```

The Geometry Pipeline

[This is preliminary documentation and subject to change.]

When you design a 3-D application, you define the world in any units you find convenient, from microns to parsecs. Your application passes a description of that world to Direct3D. This description includes the sizes and relative positions of all of the objects in your world and the position and orientation of the viewer. Direct3D transforms this description into a series of pixels on the screen. This process—the transformation of the geometry you supply into a two-dimensional image—is the geometry pipeline, sometimes called the transformation pipeline.

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

- Overview of the Pipeline
- The World Transformation
- The View Transformation
- The Projection Transformation

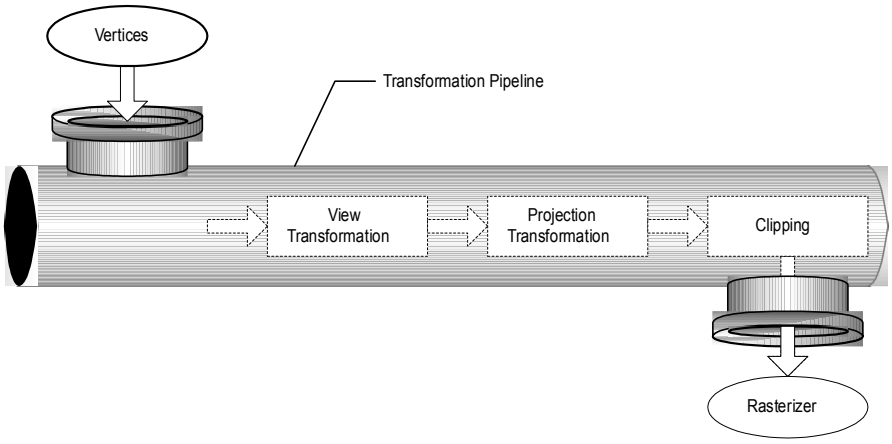
- Viewports and Clipping
- The Rasterizer

Overview of the Pipeline

[This is preliminary documentation and subject to change.]

The part of Direct3D that pushes geometry through the geometry pipeline is the transformation engine. It locates the model and viewer in the world, projects vertices for display on the screen, and clips vertices to the viewport. (The transformation engine also performs lighting computations to determine diffuse and specular components at each vertex. For more information, see Lighting and Materials.)

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



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

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

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

world space into camera space. This is the view transformation. For more information, see The View Transformation.

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

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

The World Transformation

[This is preliminary documentation and subject to change.]

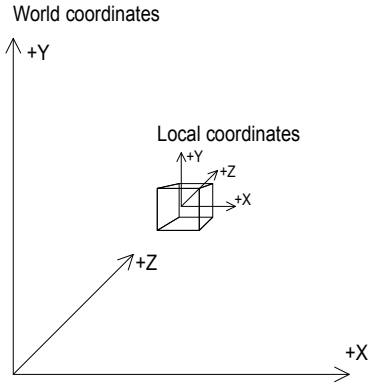
The discussion of the world transformation introduces basic concepts and provides details on how to set up a world transformation matrix in a Direct3D application. This information is organized into the following topics:

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

What Is the World Transformation?

[This is preliminary documentation and subject to change.]

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



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

Setting Up a World Matrix

[This is preliminary documentation and subject to change.]

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

The following example, from a fictitious 3-D model class, uses the helper functions in the D3dutil.cpp and D3dmath.cpp files (included with the DirectX SDK) to create a world matrix that includes three rotations to orient a model and a translation to relocate it relative to its position in world space.

```
/*
 * For the purposes of this example, the following variables
 * assumed to be valid and initialized.
 *
 * The m_vPos variable is a D3DVECTOR that contains the model's
 * location in world coordinates.
 *
 * The m_fPitch, m_fYaw, and m_fRoll variables are D3DVALUEs that
 * contain the model's orientation in terms of pitch, yaw, and roll
 * angles (in radians).
 */

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

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

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

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

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

After you prepare the world transformation matrix, call the **IDirect3DDevice3::SetTransform** method to set it, specifying the D3DTRANSFORMSTATE_WORLD flag in the first parameter. For more information, see Setting Transformations.

Performance Optimization

Direct3D uses the world and view matrices that you set through **IDirect3DDevice3::SetTransform** to configure several of its internal data structures. Each time you set a new world or view matrix, the system recalculates the associated internal structures. Setting these matrices frequently—for example, thousands of times per frame—is computationally expensive. You can minimize the number of required calculations by concatenating your world and view matrices into a proverbial "world-view" matrix that you set as the world matrix, then set the view matrix to the identity. Keep cached copies of individual world and view matrices that you can modify, concatenate, and reset the world matrix as needed. (For clarity, Direct3D samples rarely employ this optimization.)

The View Transformation

[This is preliminary documentation and subject to change.]

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

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

What Is the View Transformation?

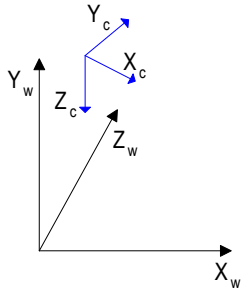
[This is preliminary documentation and subject to change.]

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

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

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

In this formula, V is the view matrix being created, T is a translation matrix that repositions objects in the world, and R_x through R_z are rotation matrices that rotate objects along the x-, y-, and z-axis. The translation and rotation matrices are based on the camera's logical position and orientation in world space. So, if the camera's logical position in the world <10,20,100>, the aim of the translation matrix is to move objects -10 units along the x-axis, -20 units along the y-axis, and -100 along the z-axis. The rotation matrices in the formula are based on the camera's orientation, in terms of how the much the axes of camera space are rotated out of alignment with world space. For example, if the camera mentioned earlier is pointing straight down, its z-axis is 90 degrees (pi/2 radians) out of alignment with the z-axis of world space, as shown in the following illustration.



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

Another approach involves creating the composite view matrix directly. (The **D3DUTIL_SetViewMatrix** helper function in the D3dutil.cpp source file uses this technique). This approach uses the camera's world space position and a "look-at point" within the scene to derive vectors that describe the orientation of the camera space coordinate axes. The camera position is subtracted from the look-at point to produce a vector for the camera's direction vector (vector *n*). Then the cross product of the vector *n* and the y-axis of world space is taken and normalized to produce a "right" vector (vector *u*). Next, the cross product of the vectors u and n is taken to determine an "up" vector (vector *v*). The right (*u*), up (*v*), and view-direction (*n*) vectors describe the orientation of the coordinate axes for camera space in terms of world space. The x, y, and z translation factors are computed by taking the negative of the dot product between the camera position and the *u*, *v*, and *n* vectors.

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

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

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

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

Setting Up a View Matrix

[This is preliminary documentation and subject to change.]

The **D3DUtil_SetViewMatrix** helper function, from the D3dutil.cpp source file that is included with this SDK, creates a view matrix based on the camera location and a look-at point passed to it. It uses the **Magnitude**, **CrossProduct**, and **DotProduct** D3D_OVERLOADS helper functions.

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

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

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

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

    D3DVECTOR vUp = vWorldUp - fDotProduct * vView;

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

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

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

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

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

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

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

    return S_OK;
}
```

As with the world transformation, you call the **IDirect3DDevice3::SetTransform** method to set the view transformation, specifying the D3DTRANSFORMSTATE_VIEW flag in the first parameter. See Setting Transformations, for more information.

Performance Optimization

Direct3D uses the world and view matrices that you set through **IDirect3DDevice3::SetTransform** to configure several of its internal data structures. Each time you set a new world or view matrix, the system recalculates the associated internal structures. Setting these matrices frequently — for example, 20000 times per frame — is computationally expensive. You can minimize the number of required calculations by concatenating your world and view matrices into a proverbial "world-view" matrix that you set as the world matrix, then set the view matrix to the identity. Keep cached copies of individual world and view matrices that you can modify, concatenate, and reset the world matrix as needed. (For clarity, Direct3D samples rarely employ this optimization.)

The Projection Transformation

[This is preliminary documentation and subject to change.]

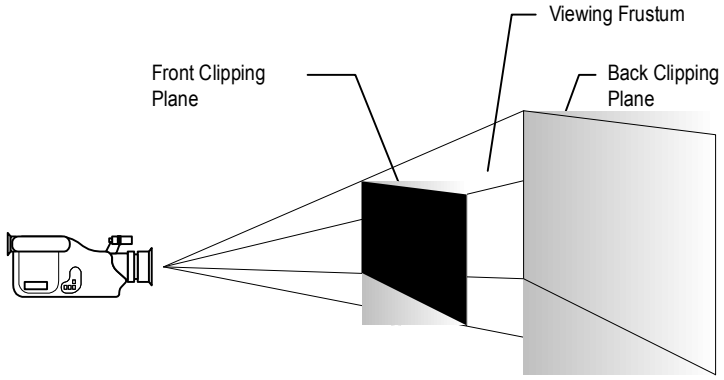
You can think of the projection transformation as controlling the camera's internals; it is analogous to choosing a lens for the camera. This is the most complicated of the three transformation types. This discussion of the projection transformation is organized into the following topics:

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

The Viewing Frustum

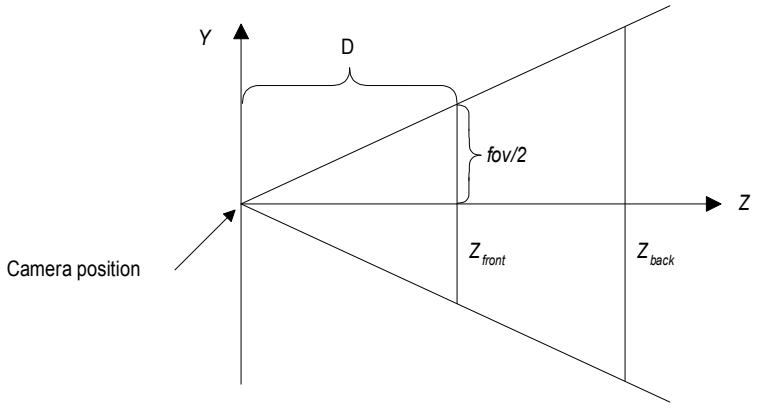
[This is preliminary documentation and subject to change.]

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



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

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



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

What Is the Projection Transformation?

[This is preliminary documentation and subject to change.]

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

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

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

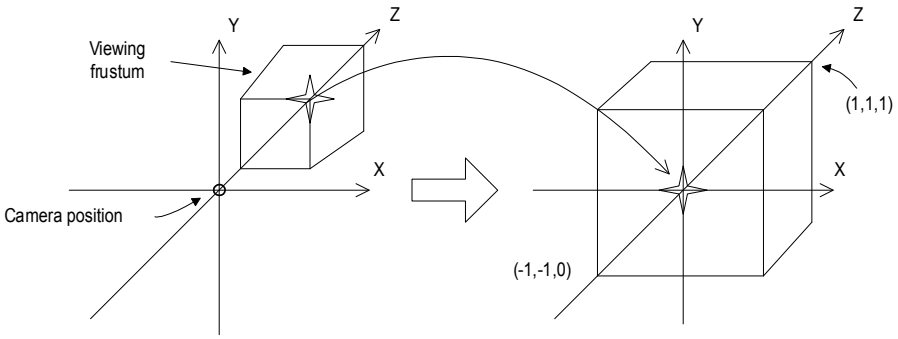
The viewing matrix puts the camera at the origin of the scene. Since the projection matrix needs to have the camera at (0, 0, -*D*), it translates the vector by -*D* in the z-direction, by using the following matrix:

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

Multiplying these two matrices gives the following composite matrix:

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

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



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

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

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

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

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

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

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

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

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

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

In these formulas, Z_n represents the position of the near clipping plane, and the V_w and V_h variables represent the width and height of the viewport, in camera space. The two dimensions correspond directly to the **dwWidth** and **dwHeight** members of the **D3DVIEWPORT2** structure.

Whatever formula you decide to use, it's important that you set Z_n to as large a value as you can, as z-values extremely close to the camera don't vary by much, making depth comparisons using 16-bit z-buffers tricky. In Direct3D, the (3,4) element of the projection matrix cannot be a negative number.

As with the world and view transformations, you call the **IDirect3DDevice3::SetTransform** method to set the projection transformation; for more information, see Setting Transformations.

Setting Up a Projection Matrix

[This is preliminary documentation and subject to change.]

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

```
D3DMATRIX
ProjectionMatrix(const float near_plane,
```

```
                    // distance to near clipping plane
                const float far_plane,
                    // distance to far clipping plane
                const float fov_horiz,
                    // horizontal field of view angle, in radians
                const float fov_vert)
                    // vertical field of view angle, in radians
{
    float    h, w, Q;

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

    D3DMATRIX ret = ZeroMatrix();
    ret(0, 0) = w;
    ret(1, 1) = h;
    ret(2, 2) = Q;
    ret(3, 2) = -Q*near_plane;
    ret(2, 3) = 1;
    return ret;
}   // end of ProjectionMatrix()
```

When you have created the matrix, you need set it in a call to the **IDirect3DDevice3::SetTransform** method, specifying D3DTRANSFORMSTATE_PROJECTION in the first parameter. For details, see Setting Transformations.

A W-Friendly Projection Matrix

[This is preliminary documentation and subject to change.]

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

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

Non-compliant

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

Compliant

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

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

Note

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

Viewports and Clipping

[This is preliminary documentation and subject to change.]

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

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

Direct3D implements clipping by way of a COM object representing a viewport's functionality and exposing interfaces to allow you to manipulate the object. The **IDirect3DViewport3** interface is the newest interface to the viewport object.

What Is a Viewport?

[This is preliminary documentation and subject to change.]

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

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

- The screen-space viewport to which the rendering will be confined.
- The background material. The viewport can be cleared to this material color.

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

The clipping volume is the area that defines the part of your scene that will appear on the render target surface. If the viewport you choose is small enough so that parts of your scene will not be visible, you should clip the parts that will be off of the screen so that the system will avoid the overhead of rendering those parts of the scene. For more information, see Clipping Volumes.

Clipping Volumes

[This is preliminary documentation and subject to change.]

The following topics discuss the role and effects of the viewport clipping volume, as well as some considerations that might apply to you, depending on the type of vertices your application uses:

- About Clipping Volumes
- Considerations for Various Vertex Types

About Clipping Volumes

[This is preliminary documentation and subject to change.]

You define a clipping volume inside the coordinate system of the viewport. The best way to define a clipping volume in Direct3D is by using the **D3DVIEWPORT2** structure—the structure that is specified by the methods in **IDirect3DViewport3**. (Another way is by using the **D3DVIEWPORT** structure. The **D3DVIEWPORT2** structure enables a better clip-volume definition than **D3DVIEWPORT** and is recommended for all DirectX 5.0 and newer applications.)

The **dvClipX**, **dvClipY**, **dvClipWidth**, and **dvClipHeight** members of the **D3DVIEWPORT2** structure specify the region inside of which vertices will be visible. This region corresponds to the destination rectangle, specified by the **dwX**, **dwY**, **dwWidth**, and **dwHeight** members of **D3DVIEWPORT2**.

Direct3D uses the values you set in the **D3DVIEWPORT2** structure to construct a matrix that it internally applies to all vertices before performing clipping tests. The matrix looks like this:

$$\begin{bmatrix} \frac{2}{C_w} & 0 & 0 & 0 \\ 0 & \frac{2}{C_h} & 0 & 0 \\ 0 & 0 & \frac{1}{Z_{max} - Z_{min}} & 0 \\ -1 - 2 \cdot \frac{C_x}{C_w} & 1 - 2 \cdot \frac{C_y}{C_h} & \frac{-Z_{min}}{Z_{max} - Z_{min}} & 1 \end{bmatrix}$$

In the preceding matrix all variables are taken directly from the **D3DVIEWPORT2** structure: C_x and C_y are the values in the **dvClipX** and **dvClipY** members, C_w and C_h are values in **dvClipWidth** and **dvClipHeight**, and the Z_{min} and Z_{max} variables are values taken from the **dvMinZ** and **dvMaxZ** members. The matrix effectively scales vertices according to the proportions of the clipping volume you define and translates them to position them around the volume's origin.

The values produced by the clipping matrix are tested by using the following formulas, and any vertices that fail the tests are clipped:

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

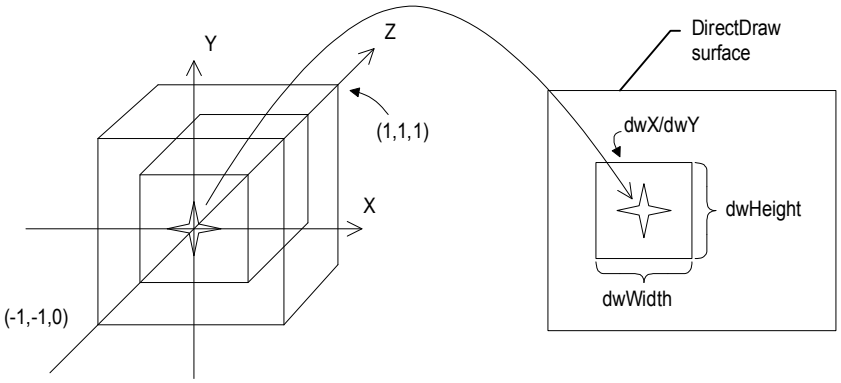
In the preceding formulas, X_c, Y_c, Z_c, and W_c represent the vertex coordinates after the clipping matrix is applied.

In most cases, you will define a clipping volume that extends from -1.0 to 1.0 in the x- and y-directions, and 0.0 to 1.0 in the z-direction, and doesn't scale vertices. The significant **D3DVIEWPORT2** structure settings for such a volume are as follows:

```
dvClipX      = -1.0;
dvClipY      =  1.0;
dvClipWidth  =  2.0;
dvClipHeight =  2.0;
dvMinZ       =  0.0;
dvMaxZ       =  1.0;
```

It is important that neither **dvClipWidth** nor **dvClipHeight** be zero. Also, **dvMinZ** cannot equal **dvMaxZ**.

This clipping volume corresponds to the destination rectangle, as shown in the following illustration.



The values you specify for the **dwX**, **dwY**, **dwWidth**, and **dwHeight** members of the **D3DVIEWPORT2** structure are screen coordinates relative to the upper-left corner of the render target surface.

Considerations for Various Vertex Types

[This is preliminary documentation and subject to change.]

If you are using **D3DVERTEX** or **D3DLVERTEX** vertices—that is, if Direct3D is performing the transformations—you might want to set the last six members of this structure as follows:

```
dvClipX = -1.0f;
dvClipY =  1.0f;
dvClipWidth = 2.0f;
dvClipHeight = 2.0f;
dvMinZ = 0.0f;
dvMaxZ = 1.0f;
```

Note that setting the viewport values as shown in the preceding example doesn't account for the viewport aspect ratio. Although it's not uncommon to use viewport parameters for aspect ratio scaling, the projection matrix is a more accurate, more flexible, and cleaner platform for the task. For more information about accounting for the viewport aspect ratio in the projection matrix, see What Is the Projection Transformation?

If you are using **D3DTLVERTEX** vertices—that is, if your application is calculating the transformations and lighting—you can set up the clip space however is best for your application. If the x- and y-coordinates in your data already match pixels, you could set the last six members of **D3DVIEWPORT2** as follows:

```
dvClipX = 0;
dvClipY = 0;
dvClipWidth = dwWidth;
dvClipHeight = dwHeight;
dvMinZ = 0.0f;
```

```
dvMaxZ = 1.0f;
```

You can use the rectangle defined by the **dwX**, **dwY**, **dwWidth**, and **dwHeight** members of the **D3DVIEWPORT2** structure to clip your geometry. Although the clipping members (their names begin with "dvClip") in **D3DVIEWPORT2** are ignored when you use **D3DTLVERTEX** vertices, the system still validates them, so you must provide reasonable values for them. If you don't need this clipping, you can specify D3DDP_DONOTCLIP in your calls to **IDirect3DDevice3::DrawPrimitive** or **IDirect3DDevice3::DrawIndexedPrimitive**.

Viewport Scaling

[This is preliminary documentation and subject to change.]

The dimensions used in the **dwX**, **dwY**, **dwWidth**, and **dwHeight** members of the **D3DVIEWPORT2** structure for a viewport define the location and dimensions of the viewport on the render target surface. These values are in screen coordinates, relative to the upper-left corner of the surface.

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

$$\begin{bmatrix} dwWidth & 0 & 0 & 0 \\ 0 & -dwHeight & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dwX & dwHeight + dwY & 0 & 1 \end{bmatrix}$$

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

Using Viewports

[This is preliminary documentation and subject to change.]

This section provides details about creating, working with, and deleting viewports. Information is divided into the following topics:

- Preparing to Use a Viewport
- Creating a Viewport
- Adding a Viewport to a Device
- Setting the Viewport Clipping Volume

- Deleting a Viewport
- Clearing a Viewport
- Manually Transforming Vertices

Preparing to Use a Viewport

[This is preliminary documentation and subject to change.]

Û To set up a viewport in Direct3D

1. Create a viewport object.

   Creating the viewport causes Direct3D to allocate internal data structures that contain the viewport's properties. For more information, see Creating a Viewport.
2. Add the viewport to a device.

   This creates an association between a viewport and a device; the device adds the viewport to an internal list of viewports. You can switch the current viewport by calling the **IDirect3DDevice3::NextViewport** method for that device. For more information about this step, see Adding a Viewport to a Device.
3. Set the viewport properties.

   After adding the viewport to a device, you can set its properties. The viewport properties control how Direct3D clips objects from a scene and determine the aspect ratio of the scene rendered to the target surface.

When you're finished working with the viewport, you can delete it from the device's viewport list. For more information, see Deleting a Viewport.

Creating a Viewport

[This is preliminary documentation and subject to change.]

Call the **IDirect3D3::CreateViewport** method to create the viewport object. For this you need a valid **LPDIRECT3D3** pointer, shown in the following example as *lpD3D2*:

```
HRESULT hr;

hr = lpD3D3->CreateViewport(&lpD3DViewport3, NULL);

if(FAILED(hr))
   return hr;
else
{
  // Add the viewport to a device.
}
```

After creating a viewport, you can add it to an existing device. For more information, see Adding a Viewport to a Device.

Adding a Viewport to a Device

[This is preliminary documentation and subject to change.]

After creating a viewport, you can call the **IDirect3DDevice3::AddViewport** method to add it to the device's viewport list. The following example shows what this call might look like:

```
HRESULT hr;

// lpD3DDevice3 is a valid pointer to a Direct3D
// device object.
hr = lpD3DDevice3->AddViewport(lpD3DViewport3);

if(FAILED(hr))
   return hr;
else
{
  // Set the viewport properties.
}
```

Once you've added the viewport to the device, you can set the viewport clipping volume. For more information, see Setting the Viewport Clipping Volume.

Setting the Viewport Clipping Volume

[This is preliminary documentation and subject to change.]

After you add a viewport to a device, you can set the viewport's clipping volume. To do this, you initialize and set clipping values for the clipping volume and for the render target surface. Viewports are commonly set up to render to the full surface and to compensate for the aspect ratio. You could use the following settings for the members of the **D3DVIEWPORT2** structure to achieve this:

```
memset(&viewData, 0, sizeof(D3DVIEWPORT2));
viewData.dwSize = sizeof(D3DVIEWPORT2);
viewData.dwX = 0;
viewData.dwY = 0;
viewData.dwWidth  = width;
viewData.dwHeight = height;
viewData.dvClipX  = -1.0f;
viewData.dvClipY  =  1.0;
viewData.dvClipWidth  = 2.0f;
viewData.dvClipHeight = 2.0f;
viewData.dvMinZ = 0.0f;     // This must be different than dvMaxZ
viewData.dvMaxZ = 1.0f;     // This must be different than dvMinZ
```

After setting values in the **D3DVIEWPORT2** structure, you apply the structure to the viewport object by calling its **IDirect3DViewport3::SetViewport2** method. The following examples shows what this call might look like:

```
HRESULT hr;

hr = lpD3DViewport3->SetViewport2(&viewData);
if(FAILED(hr))
   return hr;
```

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

The **IDirect3DViewport3** interface has two ways of specifying a viewport clipping volume. For more information, see Clipping Volumes.

Deleting a Viewport

[This is preliminary documentation and subject to change.]

When you no longer need to use a viewport with a given device, first delete any lights and materials associated with it and then remove it from the device's viewport list by calling the **IDirect3DDevice3::DeleteViewport** method. **DeleteViewport** accepts the address of an **IDirect3DViewport3** interface as its only parameter. (If you're using execute buffers, you will be calling the **IDirect3DDevice::DeleteViewport** method, which accepts the address of an **IDirect3DViewport** interface.)

Deleting a viewport only removes it from a device's viewport list; it doesn't free the resources allocated for the viewport object itself. To completely get rid of a viewport, you must release it by calling its **IUnknown::Release** method.

Clearing a Viewport

[This is preliminary documentation and subject to change.]

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

The **IDirect3DViewport3** interface offers the **IDirect3DViewport3::Clear** and **IDirect3DViewport3::Clear2** methods that provide various ways to clear the viewport. Both of the clearing methods accept one or more rectangles that define the area or areas on the surfaces being cleared. In cases where the scene being rendered includes motion throughout the entire viewport rectangle — in a first-person perspective game, perhaps — you might want to clear the entire viewport each frame. In this situation, you would set the *dwCount* parameter to 1, and the *lpRects* parameter to the address of a single rectangle that covers the entire viewport area.

Note

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

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

The clearing methods have differing features and behavior. The **IDirect3DViewport3::Clear** method can clear the viewport using the color of the background material if you specify the D3DCLEAR_TARGET flag in the *dwFlags* parameter, and it can clear the z-buffer to the "deepest" value (1.0) if you include the D3DCLEAR_ZBUFFER flag. The method requires that you set a background material by calling the **IDirect3DViewport3::SetBackground** method. The **Clear** method is incapable of clearing stencil buffers.

The **IDirect3DViewport3::Clear2** method was introduced to provide more flexibility and ease-of-use than the **Clear** method, and to provide support for clearing stencil bits within a depth buffer. **Clear2** accepts the same two flags in the *dwFlags* parameter as its ancestor, but with slightly different results, and it adds one new flag to support stencils. If you include the D3DCLEAR_TARGET flag, the method clears the viewport using an arbitrary RGBA color that you provide in the *dwColor* parameter (not the material color). If you include the D3DCLEAR_ZBUFFER flag, the method clears the depth buffer to an arbitrary depth you specify in *dvZ*: 0.0 is the closest distance, and 1.0 is the farthest. Including the new D3DCLEAR_STENCIL flag causes the method to reset the stencil bits to the value you provide in the *dwStencil* parameter. You can use integers that range from 0 to 2^n-1, where *n* is the stencil buffer bit depth.

Manually Transforming Vertices

[This is preliminary documentation and subject to change.]

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

Untransformed, unlit vertices

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

Untransformed, lit vertices

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

Transformed, lit, vertices

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

Direct3D includes two ways to change from simple to complex vertex types: the **TransformVertices** method that has been available since Direct3D was created, and vertex buffers, introduced in DirectX 6.0. The latter is the most efficient method, as vertex buffers are optimized to exploit processor specific features.

Using TransformVertices

You can use the **IDirect3DViewport3::TransformVertices** method to transform either the **D3DVERTEX** and **D3DLVERTEX** vertex types into screen coordinates, which are represented by **D3DTLVERTEX** vertices.

TransformVertices uses the current matrix (that you set with a call to the **IDirect3DDevice3::SetTransform** method) to perform the transformation. You might call **TransformVertices** if you want to transform but not necessarily render a set of vertices; for example, you might provide two vertices that define the opposite corners of a bounding box. If both vertices are clipped, you could forgo any further processing for that model.

The **TransformVertices** does not perform lighting. If you pass **D3DVERTEX** vertices, the resulting **D3DTLVERTEX** vertices will not have valid diffuse and specular components. Likewise, if you pass **D3DLVERTEX** vertices, the diffuse and specular components you provide with each vertex will be unchanged after **TransformVertices** returns.

Although managing the transformations yourself can be faster than calling the **IDirect3DViewport3::TransformVertices** method, implementing your own transformation engine can be time consuming, and **TransformVertices** is reasonably fast. If you need more control over the geometry pipeline than you get when using **D3DVERTEX** or **D3DLVERTEX** vertices, but you don't want to develop your own transformation engine, calling **TransformVertices** is a good compromise.

Note

Applications often use **TransformVertices** to perform visibility checking, because the call returns clipping information in the associated **D3DTRANSFORMDATA** structure. Although accurate, visibility checking is

best performed by calling the **IDirect3DDevice3::ComputeSphereVisibility** method, which was specially designed and optimized for this purpose.

Using Vertex Buffers

Vertex buffers are objects used to efficiently contain and process batches of vertices for rapid rendering. Vertex buffers offer the **IDirect3DVertexBuffer::ProcessVertices** method to perform vertex transformations for you; this is usually much faster than the **TransformVertices** method. The **ProcessVertices** method accepts only untransformed vertices, and can optionally light and clip vertices as well. Lighting is performed at the time you call **ProcessVertices**, but clipping is actually performed at render time.

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

The Rasterizer

[This is preliminary documentation and subject to change.]

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

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

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

Lighting and Materials

[This is preliminary documentation and subject to change.]

This section describes illumination in Direct3D Immediate Mode scenes, using ambient light, light objects, and materials. The following topics are discussed:

- Introduction to Lighting and Materials

  Describes, in general terms, the roles of lights and materials in a Direct3D Immediate Mode scene.
- The Direct3D Light Model vs. Nature

  Describes the Direct3D illumination model in more detail, and contrasts it from the behavior of light in the natural world.
- Color Values for Lights and Materials

  Describes the RGBA color values used with both lights and materials in Direct3D Immediate Mode.
- Direct Light vs. Ambient Light

  Compares and contrasts the roles of true light sources and ambient light levels.
- Enabling and Disabling the Lighting Engine

  Describes how to enable or disable the Direct3D lighting engine.
- Lights

  Describes Direct3D light objects and provides information on how to use them in a scene.
- Materials

  Describes materials in Direct3D Immediate Mode and provides information about how to use them in a scene.
- The Mathematics of Direct3D Lighting

  Provides details about the math behind the Direct3D light model.

Introduction to Lighting and Materials

[This is preliminary documentation and subject to change.]

When lighting is enabled, as Direct3D rasterizes a scene in the final stage of rendering, it determines the color of each rendered pixel based on a combination of the current material color (and the texels in an associated texture map), the diffuse and specular colors at the vertex, if specified, as well as the color and intensity of light produced by light objects in the scene or the scene's ambient light level. When you use Direct3D lighting and materials, you are allowing Direct3D to handle the details of illumination for you, but advanced users can perform lighting on their own if necessary.

How you work with lighting and materials makes a huge difference in the appearance of the rendered scene. Materials define how light reflects off of a surface. Direct light and ambient light levels define the light that is being reflected. You must use materials to render a scene if you are letting Direct3D handle lighting. Lights are not actually required to render a scene, but you'll be hard pressed to see much in a scene

rendered without light. At best, rendering an unlit scene will result in a silhouette of the objects in the scene—not enough detail for most purposes.

The Direct3D Light Model vs. Nature

[This is preliminary documentation and subject to change.]

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

Obviously, the calculations required to perfectly simulate the natural behavior of light are, by far, too time consuming to be used for real-time 3-D graphics. Therefore, with the interest of speed in mind, the Direct3D light model approximates the way light works in the natural world. Direct3D describes light in terms of red, green, and blue components that combine to create a final color. For more information, see Color Values for Lights and Materials. In Direct3D, when light reflects off a surface, the light color interacts mathematically with the surface itself to create the color eventually drawn to the screen. For specific information about the algorithms Direct3D uses, see The Mathematics of Direct3D Lighting.

The Direct3D Immediate Mode light model generalizes light into two types: ambient light and direct light. Each has different attributes, and each interacts with the material of a surface in different ways. Ambient light is light that has been scattered so much that its direction and source are indeterminate: it maintains a low-level of intensity everywhere. The indirect lighting used by photographers is a good example of ambient light. Ambient light in Direct3D, as in nature, has no real direction or source, only a color and intensity. In fact, the ambient light level is completely independent of any objects in a scene that generate light. Ambient light does not contribute to specular reflection.

Direct light is the light generated by an object in a scene; it always has color and intensity, and travels in a specified direction. Direct light interacts with the material of a surface to create specular highlights, and its direction is used as a factor in shading algorithms, including Gouraud shading. When direct light is reflected, it does not contribute to the ambient light level in a scene. The objects in a scene that generate direct light (referred to as "lights" or "light objects") have different characteristics that affect how they illuminate a scene. For more information, see Lights.

Additionally, a polygon's material has properties that affect how that polygon reflects the light it receives. You set a single reflectance trait that describes how the material reflects ambient light, and you set individual traits to determine the material's specular and diffuse reflectance. For more information, see Materials.

Color Values for Lights and Materials

[This is preliminary documentation and subject to change.]

Direct3D Immediate Mode describes color in terms the four components (red, green, blue, and alpha) that combine to make a final color. The **D3DCOLORVALUE** structure is defined to contain values for each component. Each member is a floating point value that typically ranges from 0.0 to 1.0, inclusive. Although both lights and materials use the same structure to describe color, the values within the structure are used a little differently by each.

Color values for light objects represent the amount of a particular light component it emits. Lights don't use an alpha component, so you only need to think about the red, green, and blue components of the color. You can visualize the three components as the red, green, and blue lenses on a projection television. Each lens might be off (a 0.0 value in the appropriate **D3DCOLORVALUE** structure member), it might be as bright as possible (a 1.0 value), or some level in between. The colors coming from each lens combine to make the light's final color. A combination like R: 1.0, G: 1.0, B: 1.0 creates a white light, where R: 0.0, G: 0.0, B: 0.0 results in a light that doesn't emit light at all. You can make a light that emits only one component, resulting in a purely red, green, or blue light, or, the light could use combinations to emit colors like yellow or purple. You can even set negative values color component values to create a "dark light" that actually removes light from a scene. Or, you might set the components to some value larger than 1.0 to create an extremely bright light.

With materials, on the other hand, color values represent how much of a given light component is reflected by a surface that uses that material. A material whose color components are R: 1.0, G: 1.0, B: 1.0, A: 1.0 will reflect all the light that comes its way. Likewise, a material with R: 0.0, G: 1.0, B: 0.0, A: 1.0 will reflect all of the green light that is directed at it. Materials have multiple reflectance values to create various types of effects; for more information, see Material Properties.

Color values for ambient light are different than those used for light objects and materials. For more information, see Direct Light vs. Ambient Light.

Direct Light vs. Ambient Light

[This is preliminary documentation and subject to change.]

Although both direct and ambient light illuminate objects in a scene, they are independent of one another, they have very different effects, and they require that you work with them in completely different ways.

Direct light is just that: direct. Direct light always has direction and color, it is always emitted by an object within a scene, and it is a factor for shading algorithms, such as Gouraud shading. Different types of light objects emit direct light in different ways, creating special attenuation effects. You create lights by calling the **IDirect3D3::CreateLight** method, set their properties with the **IDirect3DLight::SetLight** method, and add them to a scene by calling the **IDirect3DViewport3::AddLight** method.

Ambient light is effectively everywhere in a scene. You can think of it as a general level of light that fills an entire scene, regardless of the objects and their locations within that scene. Ambient light, being everywhere, has no position or direction, only color and intensity. Additionally, ambient light is not factored-in for shading algorithms. If you're using the DrawPrimitive architecture to do your rendering, you set the ambient light level with a single call to the **IDirect3DDevice3::SetLightState** method, specifying D3DLIGHTSTATE_AMBIENT as the *dwLightStateType* parameter, and the desired RGBA color as the *dwLightState* parameter. If you're using execute buffers, you must include the D3DOP_STATELIGHT opcode in the execute buffer, as well as the D3DLIGHTSTATE_AMBIENT flag and color value in the accompanying **D3DSTATE** structure.

Color values for ambient light are interpreted the same way for both the DrawPrimitive and execute buffer architectures. Ambient light color takes the form of an RGBA value, where each component is an integer value from 0 to 255. (This is unlike most color values in Direct3D Immediate Mode. For more information, see Color Values for Lights and Materials.) You can use the **RGBA_MAKE** macro to generate RGBA values from integers. The red, green, and blue components combine to make the final color of the ambient light. The alpha component controls the transparency of the color. In ramp emulation, ambient light doesn't have color, so the alpha component is used for brightness. When using hardware acceleration or RGB emulation, the alpha component is ignored.

Enabling and Disabling the Lighting Engine

[This is preliminary documentation and subject to change.]

Direct3D normally performs lighting calculations on any vertices that contain a vertex normal. However, you can disable lighting for vertices that include normals by using the D3DDP_DONOTLIGHT flag when you call DrawPrimitive-based rendering methods.

If your application uses vertex buffers, include or omit the D3DVOP_LIGHT flag when calling the **IDirect3DVertexBuffer::ProcessVertices** method to enable or disable lighting for that vertex buffer.

If the rendering device does not have a material assigned to it, the Direct3D lighting engine is disabled.

Lights

[This is preliminary documentation and subject to change.]

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

- Introduction to Light Objects
- Light Properties
- Using Lights

Introduction to Light Objects

[This is preliminary documentation and subject to change.]

Direct3D employs four types of lights: point lights, spotlights, directional lights, and parallel-point lights. You choose the type of light you want when you set the light properties after creating a light object. The illumination properties and the resulting computational overhead varies with each type of light. The following light types, supported in Direct3D, are discussed:

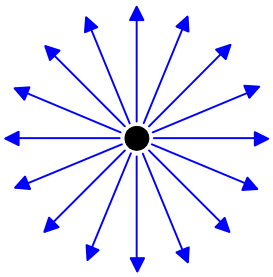
- Point lights
- Spotlights
- Directional lights
- Parallel-point lights

Do not confuse light objects with the concept of an ambient light level. For more information, see Direct Light vs. Ambient Light, Light Properties and Using Lights.

Point Lights

[This is preliminary documentation and subject to change.]

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



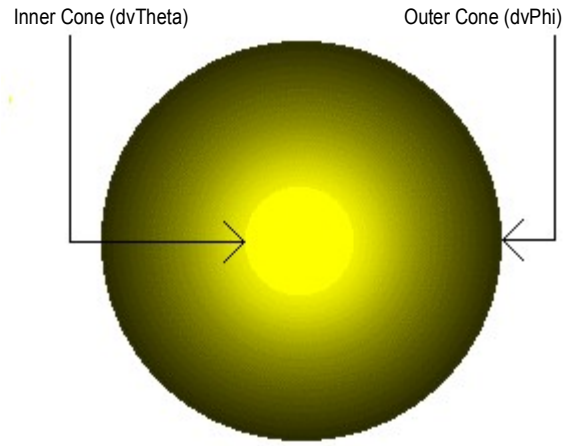
A light bulb would be a good example of a point light. Point lights are affected by attenuation and range, and illuminate a mesh on a vertex-by-vertex basis. During lighting, Direct3D uses the point light's position in world space and the coordinates of the vertex being lit to derive a vector for the direction of the light, and the distance that the light has traveled. Both of these are used (along with the vertex normal) to calculate the contribution of the light to the illumination of the surface.

Spotlights

[This is preliminary documentation and subject to change.]

Spotlights have color, position, and direction in which they emit light. Light emitted from a spotlight is made up of a bright inner cone, and a larger outer cone, with the

light intensity diminishing between the two, as shown in the following illustration, along with the related members from the **D3DLIGHT2** structure.



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

Directional Lights

[This is preliminary documentation and subject to change.]

Directional lights have only color and direction, not position. They give off parallel light, meaning that all light generated by it travels through scene in the same direction. You can imagine a directional light as a light source at near infinite distance, such as the sun. Directional lights are not affected by attenuation or range, so the direction and color you specify are the only factors considered when Direct3D calculates vertex colors. Because of the small number of illumination factors, these are the least computationally intensive lights to use.

Parallel-Point lights

[This is preliminary documentation and subject to change.]

Parallel-point lights have only color and position. They create lighting effects similar to point lights but with less overhead, at the cost of some accuracy. As a result, parallel-point lights are often a good choice when extremely accurate specular highlights are not critical. Direct3D illuminates a mesh with a parallel-point light in a manner similar to a point light, but uses a shortcut to increase performance. The shortcut taken is in establishing a single direction for the light hitting all vertices in a

mesh, rather than calculating a new direction for each vertex like a point light. (The direction that is used is the vector from the light's position to the origin of the mesh.) Because the light has the same direction for all vertices, it is considered parallel, hence the name. Like directional lights, parallel-point lights are not affected by range or attenuation.

Light Properties

[This is preliminary documentation and subject to change.]

Light properties describe a light object's type and color. Depending on the type of light being used, a light can have properties for attenuation and range, or for spotlight effects. But, not all types of lights will use all properties. Direct3D Immediate Mode uses the **D3DLIGHT2** structure to carry information about light properties for all types of lights. This section contains information for all light properties. Information is divided into the following groups:

- Light Type
- Light Color
- Light Position, Range and Attenuation
- Light Direction
- Spotlight Properties

Light properties affect how a light object illuminates objects in a scene. For more information, see Using Lights, Setting Light Properties, and The Mathematics of Direct3D Lighting.

Light Type

[This is preliminary documentation and subject to change.]

The light type property defines which type of Direct3D light object you're using. The light type is set by using a value from the **D3DLIGHTTYPE** enumeration in the **dltType** member of the light's **D3DLIGHT2** structure. There are four types of lights in Direct3D Immediate Mode — point lights, spot lights, directional lights, and parallel-point lights. Each type illuminates objects in a scene differently, with varying levels of computational overhead. For general information about how each type of light works, see Introduction to Light Objects.

Light Color

[This is preliminary documentation and subject to change.]

The color property in the **dcvColor** member of the **D3DLIGHT2** structure is an RGBA color that defines the color of light that a light object emits. The most common color is white (R:1.0 G:1.0 B:1.0), but you can create colors as needed to achieve the desired effect. For example, you could use red light for a fireplace, or you could use green light for a traffic signal set to "Go."

Generally, you set the light color components to values between 0.0 and 1.0, inclusive, but this isn't a requirement. For example, you might set all the components to 2.0, creating a light that was "brighter than white." This type of setting can be especially useful when you use attenuation settings other than constant.

Note that although Direct3D uses RGBA values for lights, the alpha color component is not used. For more information, see Color Values for Lights and Materials.

Light Position, Range and Attenuation

[This is preliminary documentation and subject to change.]

The position, range, and attenuation properties are used to define a light's location in world space, and how the light it emits behaves over distance. Like all light properties, these are carried within a light's **D3DLIGHT2** structure.

Position

Light position is described using a **D3DVECTOR** structure in the **dvPosition** member of the **D3DLIGHT2** structure. The x-, y-, and z-coordinates are assumed to be in world space. Directional lights are the only type of light that don't use the position property.

Range

A light's range property determines the distance, in world space, at which meshes in a scene no longer receive light emitted by that object. The **dvRange** member contains a floating-point value that represents the light's maximum range, in world space. Most applications set the range to the maximum possible value, D3DLIGHT_RANGE_MAX, which is defined in D3d.h. Directional and parallel-point lights don't use the range property.

Attenuation

Attenuation controls how a light's intensity decreases toward the maximum distance specified by the range property. Light attenuation is represented by three **D3DLIGHT2** structure members: **dvAttenuation0**, **dvAttenuation1**, and **dvAttenuation2**. These members contain floating point values typically ranging from 0.0 to 1.0, controlling a light's constant, linear, and quadratic attenuation. Many applications set the **dvAttenuation1** member to 1.0 and the others to 0.0, resulting in light intensity that attenuates evenly over distance — from maximum intensity at the source, to zero intensity at the light's range. You can combine attenuation values to get more complex attenuation effects. Or, you might set them to values outside of the normal range to create even stranger attenuation effects; negative attenuation values make a light that gets brighter over distance. For more information about the mathematical model that Direct3D uses to calculate attenuation, see Light Attenuation Over Distance. Like the range property, directional and parallel-point lights don't use the attenuation property.

Light Direction

[This is preliminary documentation and subject to change.]

A light's direction property determines the direction that the light emitted by the object travels, in world space. Direction is only used by directional and spotlights, and is described with a **D3DVECTOR** structure in the **dvDirection** member of the light's **D3DLIGHT2** structure. Direction vectors are described as distances from a logical origin, regardless of the light's position within a scene. Therefore, a spotlight that points straight into a scene (along the positive z-axis) would have a direction vector of <0,0,1> no matter where its position is defined to be. Similarly, you could simulate sunlight shining directly down on a scene by using a directional light whose direction is <0,-1,0>. Obviously, you don't have to create lights that shine along the coordinate axes; you can mix and match values to create lights that shine at more interesting angles.

Note

Although you don't need to normalize a light's direction vector, always be sure that it has magnitude. In other words, don't use a <0,0,0> direction vector.

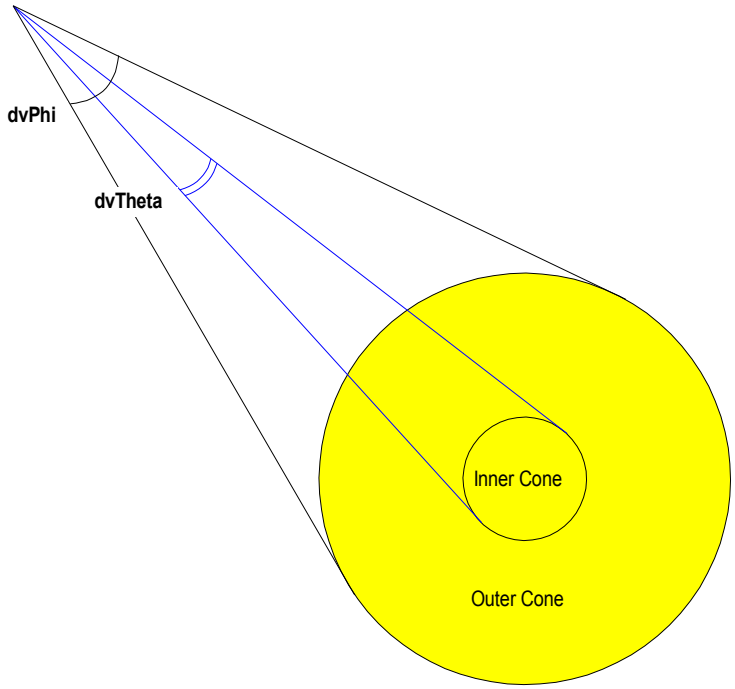
Spotlight Properties

[This is preliminary documentation and subject to change.]

The **D3DLIGHT2** structure contains three members that are used only by spotlights. These members (**dvFalloff**, **dvTheta**, and **dvPhi**) control how large or small a spotlight object's inner and outer cones are, and how light decreases between them. For general information about these characteristics, see Spotlights.

The **dvTheta** value is the radian angle of the spotlight's inner cone and the **dvPhi** value is the angle for the outer cone of light. The **dvFalloff** value controls how light intensity decreases between the outer edge of the inner cone and in the inner edge of the outer cone. Most applications will set **dvFalloff** to 1.0 to create falloff that occurs evenly between the two cones, but you can set other values as needed. For more information about the mathematical model used by Direct3D for calculating falloff, see Spotlight Falloff Model.

The following illustration shows the relationship between the values for these members, and how they can affect a spotlight's inner and outer cones of light.



Using Lights

[This is preliminary documentation and subject to change.]

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

- Preparing to Use a Light
- Creating a Light
- Setting Light Properties
- Adding a Light to a Viewport
- Deleting a Light
- Retrieving Light Properties

Preparing to Use a Light

[This is preliminary documentation and subject to change.]

The following steps are required, in order, to create and prepare a light for use in a scene. Note that these steps assume that you have already initialized the Direct3D sub-system and created a viewport.

Û To prepare a light for use in a scene

1. Create a light object.

Creating a light causes Direct3D to allocate internal data structures that will contain information about the light. For more information, see Creating a Light.

2. Set the light properties.

   By setting the light properties, you are choosing the type of light you will use, and how Direct3D will calculate lighting effects generated by that object. For more information, see Setting Light Properties.

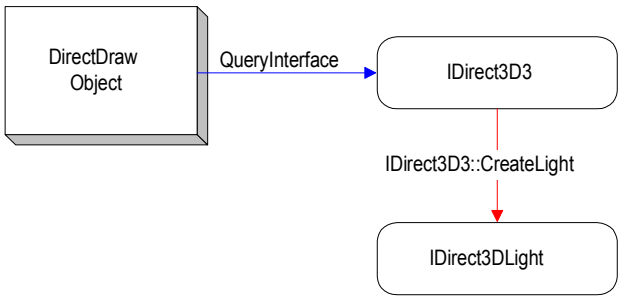
3. Add the light to a viewport.

   The final step in preparing a light is to add it to the light list for a rendering viewport. For more information, see Adding a Light to a Viewport.

Creating a Light

[This is preliminary documentation and subject to change.]

The following illustration shows a common creation path of a light in Direct3D.



You create a light object by calling the **IDirect3D3::CreateLight** method. The first parameter is the address of a variable that will contain a valid **IDirect3DLight** interface pointer if the call succeeds. The method's second parameter is intended to be used in COM aggregation — because aggregation is not implemented, this parameter must be set to NULL. The following example shows what the code to make this call might look like:

```
/*
 * For the purposes of this example, the g_lpD3D3 variable is the
 * address of an IDirect3D3 interface exposed by a Direct3D
 * object.
 */
LPDIRECT3DLIGHT g_lpD3DLight;
HRESULT   hr;

hr = g_lpD3D3->CreateLight (&g_lpD3DLight, NULL);
if (SUCCEEDED(hr))
{
   // Set the light properties.
}
else
   return hr;
```

Before you can use the light object, you must set its properties. For more information, see Setting Light Properties.

Setting Light Properties

[This is preliminary documentation and subject to change.]

After you create a light object, you must set the light's properties by preparing a **D3DLIGHT2** structure and then calling the **IDirect3DLight::SetLight** method. The **SetLight** method accepts the address of a prepared **D3DLIGHT2** structure as its only parameter. You can call **SetLight** with new information as needed to update the light's illumination properties.

The following code sets up properties for a white point light that doesn't attenuate over distance, then calls the **SetLight** method to put the properties into effect:

```
/*
 * For the purposes of this example, the g_lpD3DLight variable
 * is a valid pointer to an IDirect3D3 interface.
 */
D3DLIGHT2 g_light;
HRESULT   hr;

// Initialize the structure.
ZeroMemory(&g_light, sizeof(D3DLIGHT2));
g_light.dwSize = sizeof(D3DLIGHT2);  // MUST set the size!

// Set up for a white point light.
g_light.dltType = D3DLIGHT_POINT;
g_light.dcvColor.r = 1.0f;
g_light.dcvColor.g = 1.0f;
g_light.dcvColor.b = 1.0f;

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

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

// Make the light active light.
```

```
g_light.dwFlags = D3DLIGHT_ACTIVE;

// Set the property info for this light.
// We have to cast the LPD3DLIGHT2 to be
// an LPD3DLIGHT in order to compile.
//
// (See the following note for details.)
hr = g_lpD3DLight->SetLight((LPD3DLIGHT)&g_light);
if (SUCCEEDED(hr))
{
   // Add the light to the viewport.
}
else
   return hr;
```

Note

The **IDirect3DLight::SetLight** method checks the **dwSize** member of the structure specified in the *lpLight* parameter to determine whether you are using a **D3DLIGHT2** or **D3DLIGHT** structure. The two structures are interpreted differently. The **D3DLIGHT2** structure supersedes the **D3DLIGHT** structure, using the newer structure will provide the most reliable lighting. For backward compatibility, the **SetLight** method's parameter list is unchanged. As a result, you must cast the address of the D3DLIGHT2 structure you provide to the LPD3DLIGHT data type to avoid compiler errors.

(You can update a light's properties by a subsequent call to **SetLight** at any time, you need not add the light to the viewport each time.)

After choosing the light type by setting its properties, you can add the light to a viewport to complete the process of preparing a light for rendering. For more information, see Adding a Light to a Viewport.

Adding a Light to a Viewport

[This is preliminary documentation and subject to change.]

After you've created a light and set its properties, you can add it to a viewport by calling the viewport's **IDirect3DViewport3::AddLight** method. The **AddLight** method causes the viewport to add the light to a list of light objects it considers when rendering a scene. When you call **AddLight**, you specify the **IDirect3DLight** interface pointer that was returned when you initially created the light object by calling **IDirect3D3::CreateLight**, as shown in the following code fragment:

```
/*
 * For this example, the g_lpD3DLight variable is a pointer to
 * the IDirect3DLight interface of a light object whose
 * illumination properties have been set, and the lpViewport3
 * variable is a valid IDirect3DViewport3 interface.
```

```
 */
HRESULT hr;

hr = lpViewport3->AddLight (g_lpD3DLight);
if (FAILED (hr))
   return hr;
```

Deleting a Light

[This is preliminary documentation and subject to change.]

When you no longer need to use a light, you can delete it from a viewport by calling the viewport's **IDirect3DViewport3::DeleteLight** method. The method accepts a pointer to the **IDirect3DLight** interface of the light you want to remove.

The **DeleteLight** method only removes the light object from the list that the viewport uses when rendering a scene, it does not deallocate the light object. If you want to deallocate the light object, you must call the light object's **IUnknown::Release** method.

Retrieving Light Properties

[This is preliminary documentation and subject to change.]

You can retrieve the properties of an existing light object by calling its **IDirect3DLight::GetLight** method. When calling the **GetLight** method, you should pass the address of a **D3DLIGHT2** structure cast as the LPD3DLIGHT data type, rather than the address of a **D3DLIGHT** structure. By passing the address of a **D3DLIGHT2** structure, you ensure that there is enough memory available to accept the property information that the method copies to the provided structure from its internal data members. As with all structures in DirectX, always make sure to initialize the structure's **dwSize** member to the size of the structure, in bytes, before using it.

Materials

[This is preliminary documentation and subject to change.]

This section describes materials and how they are used in Direct3D Immediate Mode applications. The following topics are discussed:

- What are Materials?
- Material Properties
- Using Materials

What are Materials?

[This is preliminary documentation and subject to change.]

Materials describe how polygons reflect light or appear to emit light in a 3-D scene. Essentially, a material is a set of properties that tell Direct3D the following things about the polygons it is rendering:

- How they reflect ambient and diffuse light
- What their specular highlights look like
- Whether or not the polygons appear to emit light

Direct3D Immediate Mode uses the **D3DMATERIAL** structure to describe material properties. For more information, see Material Properties.

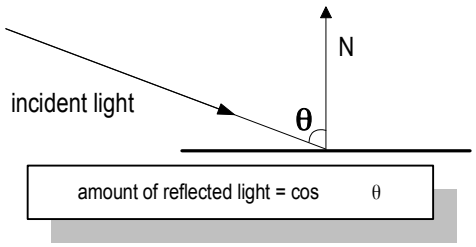
Material Properties

[This is preliminary documentation and subject to change.]

Material properties detail a material's diffuse reflection, ambient reflection, light emission, and specular highlighting characteristics. Direct3D uses the **D3DMATERIAL** structure to carry all material property information, as well as information used only for ramp emulation (an associated texture handle and the ramp palette size). Material properties affect the colors Direct3D uses to rasterize polygons that use the material. With the exception of the specular property, each of the properties is described as an RGBA color that represents how much of the red, green, and blue parts of a given type of light it reflects, and an alpha blending factor (the alpha component of the RGBA color). The material's specular property is described in two parts: color and power. For more information, see Color Values for Lights and Materials.

Diffuse and Ambient Reflection

The **dcvDiffuse** and **dcvAmbient** members of the **D3DMATERIAL** structure describe how a material reflects the ambient and diffuse light in a scene. Because most scenes contain much more diffuse light than ambient light, diffuse reflection plays the largest part in determining color. Additionally, because diffuse light is directional, the angle of incidence for diffuse light affects the overall intensity of the reflection. Diffuse reflection is greatest when the light strikes a vertex parallel to the vertex normal. As the angle increases, the effect of diffuse reflection diminishes. The amount of light reflected is the cosine of the angle between the incoming light and the vertex normal, as shown here.



Ambient reflection, like ambient light, is nondirectional. Ambient reflection has a lesser impact on the apparent color of a rendered object, but it does affect the overall

color, and is most noticeable when little or no diffuse light reflects off the material. A material's ambient reflection is affected by the ambient light set for a scene by calling the **IDirect3DDevice3::SetLightState** method with the D3DLIGHTSTATE_AMBIENT flag.

Diffuse and ambient reflection work together to determine the perceived color of an object, and are usually identical values. For example, to render a blue crystalline object, you would create a material that reflected only the blue component of diffuse and ambient light. When placed in a room with a white light, the crystal appears to be blue. However, in a room that has only red light, the same crystal would appear to be black, because its material doesn't reflect red light.

Emission

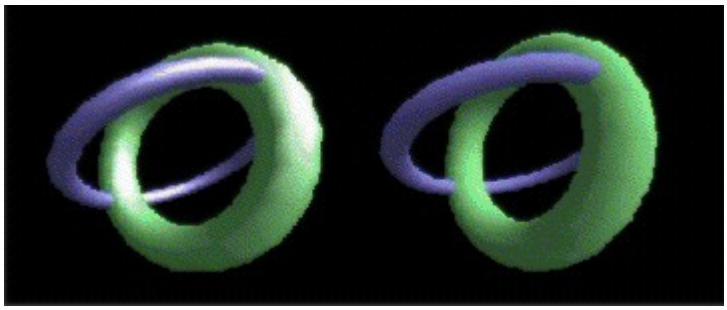
Materials can be used to make a rendered object appear to be self-luminous. The **dcvEmissive** member of the **D3DMATERIAL** structure is used to describe the color and transparency of the emitted light. Emission affects an object's color and can, for example, make a dark material brighter and take on part of the emitted color.

You can use a material's emissive property to add the illusion that an object is emitting light, without incurring the computational overhead of adding a light to the scene. In the case of the blue crystal, the emissive property could be handy if you wanted to make the crystal appear to light up, but not actually cast light on other objects in the scene. Remember, materials with emissive properties don't actually emit light that can be reflected by other objects in a scene. To achieve this effect, you would need to place an additional light within the scene.

Specular Reflection

Specular reflection creates highlights on objects, making them appear shiny. The **D3DMATERIAL** structure contains two members that describe the specular highlight color as well as the material's overall shininess. You establish the color of the specular highlights by setting the **dcvSpecular** member to the desired RGBA color — the most common colors are white or light gray. The values you set in the **dvPower** member control how sharp the specular effects are.

Specular highlights can create dramatic effects. Drawing again on the blue crystal analogy: a larger **dvPower** value will create sharper specular highlights, making the crystal appear to be quite shiny. Smaller values increase the area of the effect, creating a dull reflection that might make the crystal look frosty. To make an object truly matte, set the **dvPower** member to zero, and the color in **dcvSpecular** to black. Experiment with different levels of reflection to produce a realistic appearance for your needs. The following illustration shows two identical models, the one on the left uses a specular reflection power of 10; the model on the right has no specular reflection:



Associated Textures

When using ramp emulation, textures are assigned to materials by setting the texture's handle to the **hTexture** member in the **D3DMATERIAL** structure. You can set this member to NULL if you won't be using ramp emulation, or if no texture is associated with the material.

You can retrieve the texture handle by calling the **IDirect3DTexture2::GetHandle** method. For more information, see Textures.

Ramp Emulation Properties

The **dwRampSize** member provides information about how the material should be drawn when Direct3D uses the ramp emulation shade model for shading (as opposed to RGB emulation, or hardware acceleration). Specifically, this member is an integer value that tells Direct3D how many palette entries it should create when it renders shaded polygons. For most materials, you should set this value to a reasonable number of shades; 16 is a common value. Direct3D automatically determines the shades it uses based on the material color and number of shades you specify.

It is recommended that you use the same ramp size for all of your materials. If Direct3D runs out of palette entries when adding a new material it tries to pick the closest material already in the palette. In order to get a match, the materials must have the same ramp size. By using the same size for all materials you have a better chance of getting a good color match, providing more attractive results.

Note

A material that you will use solely as a background in a viewport should have a **dwRampSize** value of 1. This is because backgrounds are never shaded, regardless of the shade mode you set. Providing additional ramp shades for a background material doesn't provide any advantages, and effectively wastes memory.

Using Materials

[This is preliminary documentation and subject to change.]

‹Materials are required for rendering in Direct3D Immediate Mode. › This section contains information about using Materials in a Direct3D application. Information is divided into the following topics:

- Preparing to Use a Material
- Creating a Material
- Setting Material Properties
- Retrieving Material Handles
- Selecting a Material for Rendering
- Retrieving Material Properties

Preparing to Use a Material

[This is preliminary documentation and subject to change.]

The following steps are required, in order, to create and prepare a material for rendering.

Û To prepare a material for rendering

1. Create a material object.

   When you create a material object, Direct3D allocates internal data structures that will contain information about the material. For more information, see Creating a Material.

2. Set the material properties.

   By setting material properties, you define how the material will reflect light in a scene, and how material colors will be used when Direct3D is using ramp emulation. For more information, see Setting Material Properties.

3. Retrieve the material handle.

   Retrieving the material object's handle creates an association between the material and a Direct3D device. For more information, see Retrieving Material Handles.

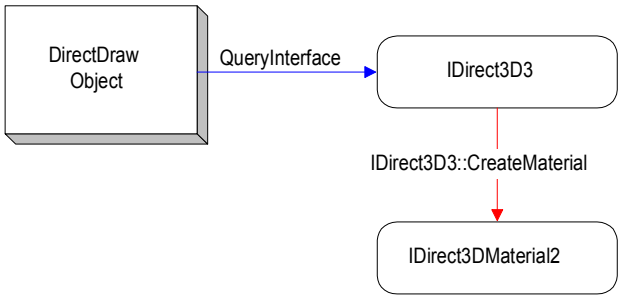
4. Select the material.

   Selecting the material causes Direct3D to rasterize polygons with that material's reflection properties, and is the final step you must take to prepare the material for rendering. For more information, see Selecting a Material for Rendering.

Creating a Material

[This is preliminary documentation and subject to change.]

The following illustration shows a common creation path of a material in Direct3D.



You create a material by calling the **IDirect3D3::CreateMaterial** method. The first parameter is the address of the variable that will contain a valid **IDirect3DMaterial3** interface pointer if the call succeeds. The method's second parameter is unused and should be set to NULL. The following example shows how this call can be made:

```
//
// The g_lpD3D variable is a global variable that
// contains a pointer to an IDirect3D3 interface.
//

LPDIRECT3DMATERIAL3 lpMat3;
HRESULT     hr;

hr = g_lpD3D->CreateMaterial(&lpMat3, NULL);
if(SUCCEEDED(hr))
{
   // Set the material properties.
}
```

Before using the material, you must set its properties. For more information, see Setting Material Properties.

Setting Material Properties

[This is preliminary documentation and subject to change.]

After creating a material object, you must set its properties before you can use it during rendering. Setting the material's properties involves preparing a **D3DMATERIAL** structure and then calling the **IDirect3DMaterial3::SetMaterial** method.

To prepare the **D3DMATERIAL** structure for use, set the property information in the structure to create the desired effect during rendering. The following code fragment sets up the **D3DMATERIAL** structure for a purple material, with sharp white specular highlights, and a 16 color ramp (which Direct3D only uses for ramp emulation):

```
D3DMATERIAL mat;
```

```
// Initialize the structure for use.
ZeroMemory(&mat, sizeof(D3DMATERIAL));
mat.dwSize   = sizeof(D3DMATERIAL);  // This is REQUIRED.

// Set the RGBA for diffuse reflection.
mat.dcvDiffuse.r = (D3DVALUE)0.5;
mat.dcvDiffuse.g = (D3DVALUE)0.0;
mat.dcvDiffuse.b = (D3DVALUE)0.5;
mat.dcvDiffuse.a = (D3DVALUE)1.0;

// Set the RGBA for ambient reflection.
mat.dcvAmbient.r = (D3DVALUE)0.5;
mat.dcvAmbient.g = (D3DVALUE)0.0;
mat.dcvAmbient.b = (D3DVALUE)0.5;
mat.dcvAmbient.a = (D3DVALUE)1.0;

// Set the color and sharpness of specular highlights.
mat.dcvSpecular.r = (D3DVALUE)1.0;
mat.dcvSpecular.g = (D3DVALUE)1.0;
mat.dcvSpecular.b = (D3DVALUE)1.0;
mat.dcvSpecular.a = (D3DVALUE)1.0;
mat.dvPower = (float)50.0;

// Use a 16 entry color ramp, in case
// we're using ramp emulation.
mat.dwRampSize = 16;
```

After preparing the **D3DMATERIAL** structure, complete setting material properties by calling the **IDirect3DMaterial3::SetMaterial** method of the desired material object. This method accepts the address of a prepared **D3DMATERIAL** structure as its only parameter. You can call **SetMaterial** with new information as needed to update the material's illumination properties.

Once you successfully set a material's properties, you can retrieve its material handle, which you need to select the material during rendering. For more information, see Retrieving Material Handles.

Retrieving Material Handles

[This is preliminary documentation and subject to change.]

After setting a material object's properties, you can retrieve its material handle by calling the **IDirect3DMaterial3::GetHandle** method. Direct3D Immediate Mode uses the **D3DMATERIALHANDLE** data type to declare material handle variables.

By retrieving the material handle, you are effectively creating an association between the material and a particular Direct3D device. The material handle represents that

association — the handle you retrieve can only be used with that device. To use the material with another device, you must retrieve another handle specific to that device.

The **GetHandle** method accepts two parameters: the address of an **IDirect3DDevice3** interface, and the address of a variable that will contain the resulting material handle after the call returns. Upon retrieving the material's handle, you can use it during rendering as often as you need. For more information, see Selecting a Material for Rendering.

Selecting a Material for Rendering

[This is preliminary documentation and subject to change.]

After you've created a material object, set its properties, and retrieved its handle, you're ready to render. You select the material into a Direct3D device by calling the device's **IDirect3DDevice3::SetLightState** method with the appropriate flags. The **SetLightState** method is a multi-purpose method; when calling the method to select the rendering material, set the first parameter to D3DLIGHTSTATE_MATERIAL, and set the second parameter to the handle of the material you want to use. The following code shows what this call commonly looks like

```
//
// For this example, hMat is a variable of type D3DMATERIALHANDLE that has
// been set to a valid material handle.  The lpDev3 variable is a pointer
// to the IDirect3DDevice3 interface of the device that will use the material.
//
HRESULT hr;

// Set the current material.
hr = lpDev3->SetLightState(D3DLIGHTSTATE_MATERIAL, hMat);
if(FAILED(hr))
{
   REPORTERR(hr);
   return hr;
}
```

Note

A Direct3D device can render with only one material at a time. After setting the current material, Direct3D will use that material for all polygons until another material is selected. You can change the currently used material at any time.

Retrieving Material Properties

[This is preliminary documentation and subject to change.]

You retrieve a material object's properties by calling the object's **IDirect3DMaterial3::GetMaterial** method. Unlike using the **IDirect3DMaterial3::SetMaterial** method, **GetMaterial** doesn't require much preparation. Before the call, initialize a **D3DMATERIAL** structure's members to

zero and set the **dwSize** member to the size of the structure, in bytes. The **GetMaterial** method accepts the address of an initialized **D3DMATERIAL** structure, and fills the provided structure with information describing the current material properties before returning.

The Mathematics of Direct3D Lighting

[This is preliminary documentation and subject to change.]

Direct3D models illumination by estimating how light behaves in nature. The Direct3D light model keeps track of light color, the direction and distance that light travels, the position of the viewer, and the characteristics of materials to compute two color components for each vertex in a face.

Note

All computations are made in model space by transforming the light's position and direction, along with the camera position to model space using the inverse of the world matrix, then backtransformed. Direct3D uses these color components to compute the color it draws while rasterizing the pixels of a face. As a result, if the world or view matrices introduce non-uniform scaling, the resultant lighting could be inaccurate.

This section presents a technical look at the formulas that Direct3D uses to come up with diffuse and specular components. By understanding Direct3D's approach, you will be better equipped to decide if the Direct3D light model suits your needs. The Direct3D light model was designed to be accurate, efficient, and easy to use. However, if the formulas used by Direct3D don't suit your needs, you can implement your own light model, bypassing the Direct3D lighting module altogether.

- Light Attenuation Over Distance
- Reflectance Model
- Spotlight Falloff Model

Light Attenuation Over Distance

[This is preliminary documentation and subject to change.]

Direct3D uses the following formula to normalize the distance from a light source to a vertex into a value from 0.0 to 1.0, inclusive:

$$D_n = \frac{R - D}{R}$$

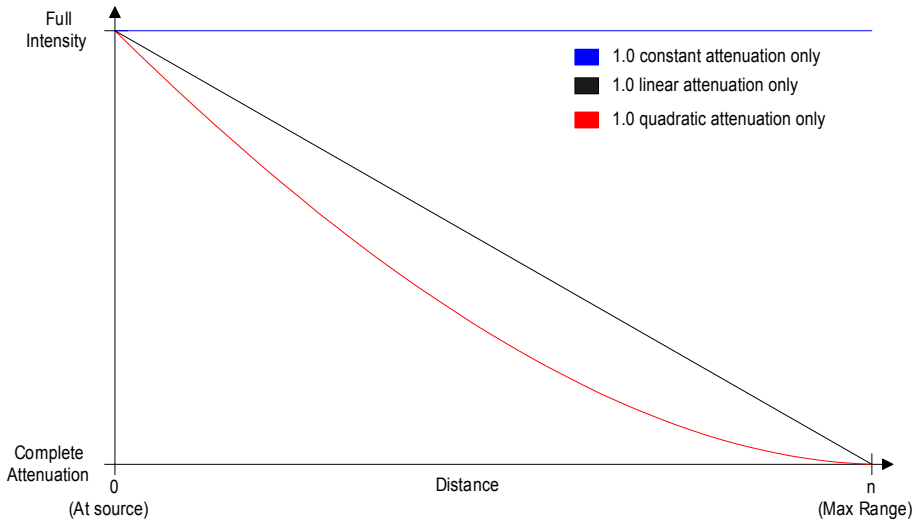
In the preceding formula, D_n is the normalized distance, R is the light's range, and D is the distance, in world space, from the light source to the vertex being lit. (When D is greater than R, the system assumes that no light reaches the vertex, and it moves to the next vertex to be lit.) A normalized distance is 1.0 at the light's source, and 0.0 at the light's range.

With the normalized distance in hand, Direct3D then applies the following formula to calculate light attenuation over distance for point lights and spotlights (directional and parallel point lights don't attenuate over distance):

$$A = dvAttenuation0 + D_n \times dvAttenuation1 + D_n^2 \times dvAttenuation2$$

In this attenuation formula, A is the calculated total attenuation and D_n is the normalized distance from the light source to the vertex. The *dvAttenuation0*, *dvAttenuation1*, and *dvAttenuation2* values are the light's constant, linear, and quadratic attenuation factors as specified by the members of a light object's **D3DLIGHT2** structure. (Not surprisingly, the corresponding structure members are **dvAttenuation0**, **dvAttenuation1**, and **dvAttenuation2**. In most cases, these attenuation factors are between 0.0 and 1.0, inclusive.)

The constant, linear and quadratic attenuation factors act as coefficients in the formula — you can produce a wide variety of attenuation curves by making simple adjustments to them. Most applications will set the linear attenuation factor to 1.0 and set the remaining factors to 0.0 to produce a light that steadily falls off over distance. Similarly, you could apply a constant attenuation factor of 1.0 by itself to make a light that doesn't attenuate (but will still be limited by range). The following illustration shows the three most common attenuation curves.



Note that using only quadratic attenuation often results in harsh transition between light and dark over distance. Each of the curves uses only one of the attenuation factors, but you are free to mix these values to create different attenuation effects.

The attenuation formula used by Direct3D computes an attenuation value that typically ranges from 1.0 at the light source to 0.0 at the maximum range of the light. (The result of the formula isn't normalized to fit between 0.0 and 1.0; attenuation outside that range is still considered valid, but will result in harsh or unpredictable lighting). The attenuation value is multiplied into the red, green and blue components of the light's color to scale the light's intensity as a factor of the distance light travels

to a vertex. After computing the light attenuation, Direct3D also considers spotlight effects (if applicable), the angle that the light reflects from a surface, as well as the reflectance of the material that the vertex uses to come up with the diffuse and specular components for that vertex. For more information, see Spotlight Falloff Model and Reflectance Model.

Reflectance Model

[This is preliminary documentation and subject to change.]

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

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

This section provides information on the methods that the system uses for calculating reflectance. Information is divided according to the type of reflectance being calculated:

- Diffuse Reflection Model
- Specular Reflection Model

Diffuse Reflection Model

[This is preliminary documentation and subject to change.]

Direct3D uses the following formula to compute diffuse reflection factors:

$$R_d = -D \bullet N$$

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

After the diffuse reflection formula is applied, the light has been scaled appropriately for attenuation over distance, spotlight effects, and diffuse reflection. The scaled light is then applied to the diffuse reflectance property of the material that the vertex uses and ambient reflection is considered to determine the diffuse component at that vertex. The formula that combines ambient and diffuse reflection to create the diffuse component for the vertex looks like this:

$$D_v = I_a M_a + M_e + A R_d C_d$$

In the preceding formula, D_v is the diffuse component being calculated for the vertex, I_a is the ambient light level in the scene, M_a is the material's ambient reflection property, and the M_e variable is the emissive property for the material. The A variable is the attenuated light at the vertex (see Light Attenuation Over Distance) and R_d is the diffuse reflectance factor. The C_d variable can be one of two possible colors, the one used depends on the state of the system and the format of the vertex at the time. If the D3DLIGHTSTATE_COLORVERTEX light state is enabled (and diffuse color is present in the vertex), the system uses the diffuse vertex color in C_d. Otherwise, the system uses the diffuse material color for C_d. If a diffuse vertex color is present, the output alpha is equal to the diffuse alpha for the vertex. Otherwise, output alpha is equal to the alpha component of diffuse material, clamped to the range [0, 255].

After applying this formula, D_v is the diffuse color component for the vertex being lit.

For more information, see Specular Reflection Model, Spotlight Falloff Model.

Specular Reflection Model

[This is preliminary documentation and subject to change.]

Modeling specular reflection requires that the system not only know the direction that light is traveling, but also the direction to the viewer's eye. Direct3D uses the inverses of the view- and world-transformation matrices to get this information. The system uses a simplified version of the Phong specular-reflection model, which employs a "halfway vector" that exists midway between the vector to the light source and the vector to the eye to approximate the intensity of specular reflection.

Following the simplified Phong model, Direct3D determines the halfway vector by subtracting the vector to the light source from the vector to the eye. Once it has found the halfway vector, the system uses the following formula to compute specular reflection:

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

In the preceding formula, R_s is the specular reflectance, N in the vertex normal, H is the halfway vector, and p is the specular reflection power of the material that the vertex uses (as specified by the **dvPower** member of the material's **D3DMATERIAL** structure). The N and H vectors are normalized.

Like the diffuse reflectance formula, this formula produces values that range from -1.0 to 1.0, which are clamped to the range of 0.0 to 1.0 and used to scale the light reflecting from the vertex. Also similar to the diffuse reflection model, the remaining light is applied to the specular reflectance property of the vertex's material to derive the specular component at that vertex, as shown in the following formula:

$$S_v = A R_s M_s$$

In the preceding formula, S_v is the specular color being computed, A is the attenuated light at the vertex, and the R_s variable is the previously calculated specular reflectance. The C_s variable can be one of two possible colors, the one used depends on the state of the system and the format of the vertex at the time. If the D3DLIGHTSTATE_COLORVERTEX light state is enabled (and specular color is present in the vertex), the system uses the specular vertex color in C_s. Otherwise, the system uses the diffuse material color for C_s.

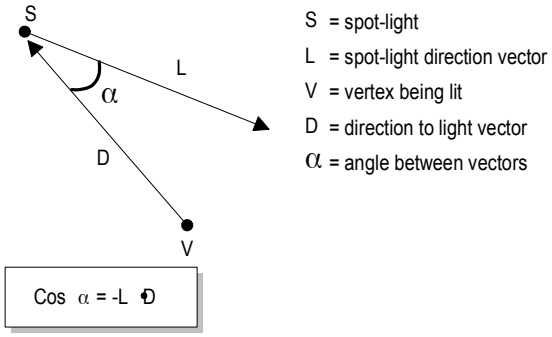
For more information, see Diffuse Reflection Model.

Spotlight Falloff Model

[This is preliminary documentation and subject to change.]

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

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



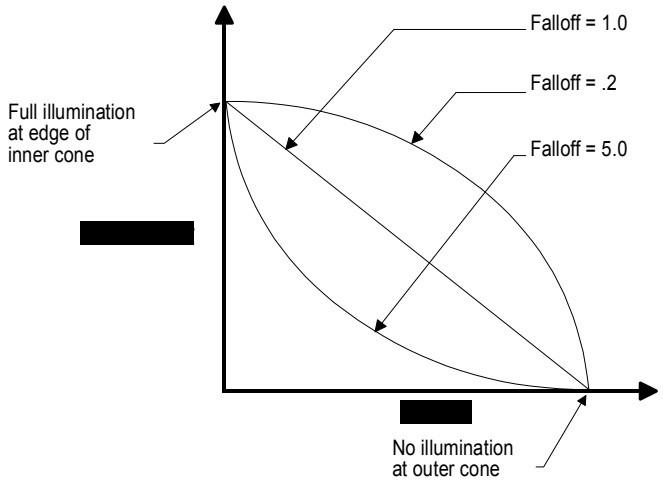
Next, the system compares this value to the cosine of the spot light's inner and outer cone angles. In the light's **D3DLIGHT2** structure, the **dvTheta** and **dvPhi** members represent the total cone angles for the inner and outer cones. Because the attenuation occurs as the vertex becomes more distant from the center of illumination, rather than across the total cone angle, Direct3D halves these cone angles before calculating their cosines.

If the dot product of vectors L and D is less than or equal to the cosine of the outer cone angle, the vertex lies beyond the outer cone, and receives no light. If the dot product of L and D is greater than the cosine of the inner cone angle, then the vertex is within the inner cone, and receives the maximum amount of light (still considering

attenuation over distance). If the vertex is somewhere between the two regions, Direct3D calculates falloff for the vertex by using the following formula:

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

In the formula, I_f is light intensity (after falloff) for the vertex being lit, α is the angle between vectors L and D, ϕ is half of the outer cone angle, θ is half of the inner cone angle, and p is the spot light's falloff property (**dvFalloff** in the **D3DLIGHT2** structure). This formula generates a value between 0.0 and 1.0 that scales the light's intensity at the vertex to account for falloff. Attenuation as a factor of the vertex's distance from the light is also applied. The p value corresponds to the **dvFalloff** member of the **D3DLIGHT2** structure and controls the shape of the falloff curve. The following illustration shows how different **dvFalloff** values can affect the falloff curve:



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

For more information, see Light Attenuation Over Distance.

Vertex Formats

[This is preliminary documentation and subject to change.]

This section describes the concepts you need to understand to specify vertices in Direct3D, and provides information about the various formats your application can use to declare vertices. The following topics are discussed:

- About Vertex Formats

- Untransformed and Unlit Vertices
- Untransformed and Lit Vertices
- Transformed and Lit Vertices
- Strided Vertex Format

Note

Prior to DirectX 6.0, applications were required to use one of three vertex types —**D3DVERTEX**, **D3DLVERTEX**, and **D3DTLVERTEX**—depending on which parts of the Direct3D geometry pipeline were being used. With the introduction of more flexible vertex formats in DirectX 6.0, you can declare vertices in many more ways than before, but you can still use the predefined structures to describe untransformed and unlit vertices, untransformed but lit vertices, and vertices that are both transformed and lit. For more information, read the topics for each type of vertex in this section thoroughly.

About Vertex Formats

[This is preliminary documentation and subject to change.]

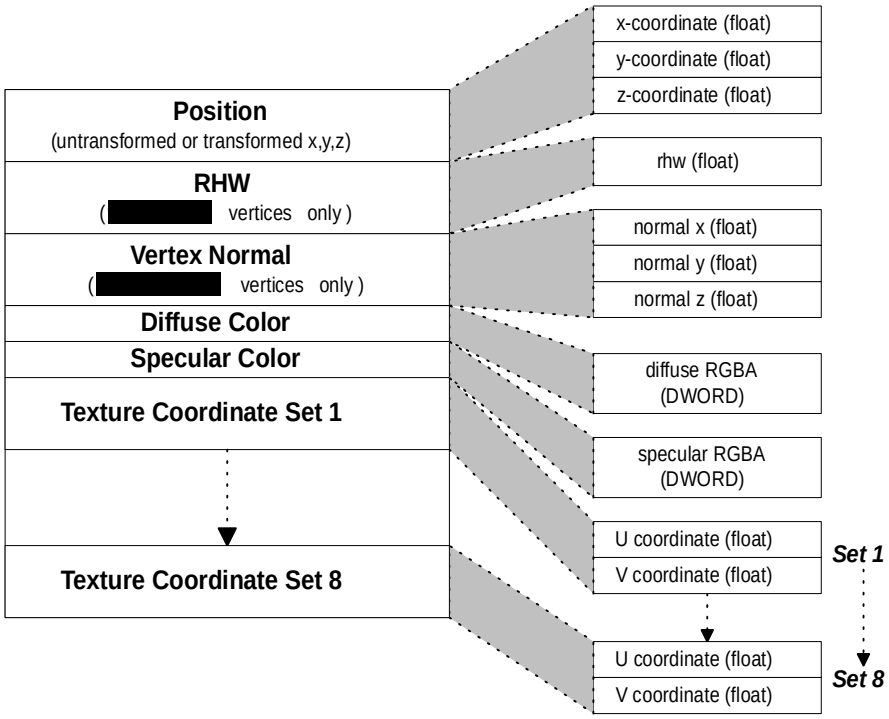
Direct3D Immediate Mode applications can define model vertices in several different ways. Support for flexible vertex definitions (also known as "flexible vertex formats") makes it possible for your application to use only the vertex components it needs, eliminating those components that aren't used. By using only the needed vertex components, your application can conserve memory and minimize the processing bandwidth required to render models.

You describe how your vertices are formatted by using a combination of flexible vertex format flags. Each of the rendering methods of **IDirect3DDevice3** accepts a combination of these flags, and uses them to determine how to render primitives. Basically, these flags tell the system which vertex components — position, normal, colors, and the number of texture coordinates — your application uses and, indirectly, which parts of the rendering pipeline you want Direct3D to apply to them. In addition, the presence or absence of a particular vertex format flag communicates to the system which vertex component fields are present in memory, and which you've omitted.

Note

The behavior is slightly different if your application uses the **IDirect3DDevice3::DrawPrimitiveStrided** or **IDirect3DDevice3::DrawIndexedPrimitiveStrided** methods. For more information, see Strided Vertex Format.

One significant requirement that the system places on how you format your vertices is on the order in which the data appears. The following illustration depicts the required order for all possible vertex components in memory, and their associated data types.



No real application will use every single component — the RHW (reciprocal homogenous W) and vertex normal fields are mutually exclusive–nor will most applications try to use all eight sets of texture coordinates, but the flexibility is there. There are several restrictions on which flags you can use with other flags. These are mostly common-sense. For example, you can't use the D3DFVF_XYZ and D3DFVF_XYZRHW flags together, as this would indicate that your application is describing a vertex's position with both untransformed and transformed vertices. For more information, take a look at the description for each of the flags in Flexible Vertex Format Flags.

Untransformed and Unlit Vertices

[This is preliminary documentation and subject to change.]

The presence of the D3DFVF_XYZ and D3DFVF_NORMAL flags in the vertex description that you pass to rendering methods identifies the untransformed and unlit vertex type. By using untransformed and unlit vertices, your application effectively requests that Direct3D perform all transformation and lighting operations using its internal algorithms. (If you want, you can pass D3DDP_DONOTLIGHT to the rendering methods to disable Direct3D's lighting engine for the primitives being rendered.)

Most applications use this vertex type, as it frees them from implementing their own transformation and lighting engines. However, because the system is making

calculations for you, it requires that you provide a certain amount of information with each vertex:

- You are required to specify vertices in untransformed model coordinates (a model coordinate vertex is positioned relative to a local origin for the model, not for the world). The system then applies world, view, and projection transformations to the model coordinates to position them within your scene, and determine their final locations on the screen.
- You should include a vertex normal. Omitting the vertex normal is like telling Direct3D that you've lit the vertices already (see Untransformed and Lit Vertices). The system uses the vertex normal, along with the current material, in its lighting calculations. For details, see Face and Vertex Normal Vectors, and Lighting and Materials.

Other than these requirements, you have the flexibility to use (or disregard) the other vertex components. For example, if you want to include a diffuse or specular color with your untransformed vertices, you can. (This wasn't possible before DirectX 6.0). These color components can "tint" the material color at each vertex, making it possible to achieve shading effects that are much more subtle and flexible than lighting calculations that use only the material color. Untransformed, unlit vertices can also include up to eight sets of texture coordinates.

Applications can still use the legacy **D3DVERTEX** structure for vertices. In fact, the d3dtypes.h header file defines a shortcut macro to identify this vertex format:

```
#define D3DFVF_VERTEX ( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1 )
```

If the **D3DVERTEX** structure doesn't suit your application's needs, feel free to define your own. Remember which vertex components your application needs, and make sure they appear in the required order by declaring a properly ordered structure. The following code declares a valid vertex format structure that includes a position, a vertex normal, a diffuse color, and two sets of texture coordinates:

```
//
// The vertex format description for this vertex
// would be: (D3DFVF_XYZ | D3DFVF_NORMAL |
//           D3DFVF_DIFFUSE | D3DFVF_TEX2)
//
typedef struct _UNLITVERTEX {
    float x, y, z;       // position

    float nx, ny, nz;    // normal

    DWORD dwDiffuseRGBA;  // diffuse color

    float tu1,    // texture coordinates
          tv1;
```

```
    float tu2,
          tv2;
} UNLITVERTEX, *LPUNLITVERTEX;
```

The vertex description for the preceding structure would be a combination of the D3DFVF_XYZ, D3DFVF_NORMAL, D3DFVF_DIFFUSE, and D3DFVF_TEX2 flexible vertex format flags. The rendering methods, such as **IDirect3DDevice3::DrawPrimitive**, accept the address of a vertex array as a void pointer, so remember to cast your vertex array pointer to the **LPVOID** data type when you call the rendering methods.

For more information, see About Vertex Formats.

Untransformed and Lit Vertices

[This is preliminary documentation and subject to change.]

If you include the D3DFVF_XYZ flag, but not the D3DFVF_NORMAL flag, in the vertex format description you use with the Direct3D rendering methods, you are identifying your vertices as untransformed, but already lit. (For information about other dependencies and exclusions, see Flexible Vertex Format Flags.)

By using untransformed and lit vertices, your application requests that Direct3D not perform any lighting calculations on your vertices, but it should still transform them using the previously set world, view, and projection matrices. Because the system isn't doing lighting calculations, it doesn't need a vertex normal. The system uses the diffuse and specular components at each vertex for shading. These colors might be arbitrary, or they might be computed using your own lighting formulas. If you don't include a diffuse or specular component, the system uses the default colors. The default diffuse color is 0xFFFFFFFF, and the default specular color is 0x00000000.

Like the other vertex types, other than including a position and some amount of color information, you are free to include or disregard the texture coordinate sets in the unlit vertex format.

Applications can still use the legacy **D3DLVERTEX** structure for vertices. The d3dtypes.h header file defines the following helper macro that you can use to describe the **D3DLVERTEX** structure's format:

```
#define D3DFVF_LVERTEX ( D3DFVF_XYZ | D3DFVF_RESERVED1 | D3DFVF_DIFFUSE | \
                         D3DFVF_SPECULAR | D3DFVF_TEX1 )
```

Note that the helper macro includes the D3DFVF_RESERVED1 flag, indicating to the system that you're using the **D3DLVERTEX** structure, which includes the **dwReserved** member. This is required when using **D3DLVERTEX** because vertex formats don't usually include reserved fields; the D3DFVF_RESERVED1 flag informs the system that there is an unused **DWORD** between the vertex's position and diffuse color vertex components.

If the **D3DLVERTEX** structure doesn't include all the fields your application needs, you can define another structure. Make sure that your vertex components appear in the required order, declaring a new structure accordingly. The following code declares a valid untransformed and lit vertex, with diffuse and specular vertex colors, and three sets of texture coordinates:

```
//
// The vertex format description for this vertex
// would be: (D3DFVF_XYZ | D3DFVF_DIFFUSE |
//           D3DFVF_SPECULAR | D3DFVF_TEX3)
//
typedef struct _LITVERTEX {
    float x, y, z;        // position

    DWORD dwDiffuseRGBA;  // diffuse color

    DWORD dwSpecularRGBA; // specular color

    float tu1,    // texture coordinates
          tv1;

    float tu2,
          tv2;

    float tu3,
          tv3;
} LITVERTEX, *LPLITVERTEX;
```

The vertex description for the preceding structure would be a combination of the D3DFVF_XYZ, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX3 flexible vertex format flags. The rendering methods, such as **IDirect3DDevice3::DrawPrimitive**, accept the address of a vertex array as a void pointer, so remember to cast your vertex array pointer to the **LPVOID** data type when you call the rendering methods.

For more information, see About Vertex Formats.

Transformed and Lit Vertices

[This is preliminary documentation and subject to change.]

If you include the D3DFVF_XYZRHW flag in your vertex format description, you are telling the system that your application uses transformed and lit vertices. This means that Direct3D doesn't transform your vertices with the world, view, or projection matrices, nor does it perform any lighting calculations; it assumes that your application has already taken care of these steps. (This fact makes transformed and lit vertices common when porting existing 3-D applications to Direct3D Immediate

Mode.) In short, Direct3D does not modify transformed and lit vertices at all; it passes them directly to the driver to be rasterized.

The vertex format flags associated with untransformed vertices and lighting (D3DFVF_XYZ and D3DFVF_NORMAL) are not allowed if D3DFVF_XYZRHW is present. For more about flag dependencies and exclusions, see Flexible Vertex Format Flags.

The system requires that the vertex position you specify be already transformed. The x and y values must be in screen coordinates, and z must be the depth value of the pixel to be used in the z-buffer. Z values can range from 0.0 to 1.0, where 0.0 is the closest possible position to the viewer, and 1.0 is the farthest position still visible within the viewing area. Immediately following the position, transformed and lit vertices must include an RHW value (reciprocal of homogeneous W) value. RHW is the reciprocal of the W coordinate from the homogeneous point (x,y,z,w) at which the vertex exists in *projection space*. (This value often works out to be the distance from the eyepoint to the vertex, taken along the z-axis.)

Other than the position and RHW requirements, this vertex format is similar to an untransformed and lit vertex. To recap:

- The system doesn't do any lighting calculations with this format, so it doesn't need a vertex normal.
- You can specify a diffuse or specular color. If you don't, the system uses 0xFFFFFFFF and 0x00000000 for these components, respectively.
- You can use up to eight sets of texture coordinates, or none at all.

Applications can still use the legacy **D3DTLVERTEX** structure for transformed and lit vertices. The d3dtypes.h header file defines the following helper macro that you can use to describe the vertex format declared by the **D3DTLVERTEX** structure:

```
#define D3DFVF_TLVERTEX ( D3DFVF_XYZRHW | D3DFVF_DIFFUSE |
D3DFVF_SPECULAR | \
                    D3DFVF_TEX1 )
```

It's possible that the **D3DTLVERTEX** structure doesn't include the fields you need. If this is the case, define another structure that does, but make sure that the vertex components are ordered properly. The following code declares a valid transformed and lit vertex, with diffuse and specular vertex colors, and one set of texture coordinates:

```
//
// The vertex format description for this vertex
// would be: (D3DFVF_XYZRHW | D3DFVF_DIFFUSE |
//          D3DFVF_SPECULAR | D3DFVF_TEX1)
//
typedef struct _TRANSLITVERTEX {
   float x, y;          // screen position
```

```
    float z;            // Z-buffer depth

    float rhw;          // reciprocal homogeneous W

    DWORD dwDiffuseRGBA;  // diffuse color

    DWORD dwSpecularRGBA; // specular color

    float tu1,    // texture coordinates
          tv1;
} TRANSLITVERTEX, *LPTRANSLITVERTEX;
```

The vertex description for the preceding structure would be a combination of the D3DFVF_XYZRHW, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX1 flexible vertex format flags. The rendering methods, such as **IDirect3DDevice3::DrawPrimitive**, accept the address of a vertex array as a void pointer, so remember to cast your vertex array pointer to the **LPVOID** data type when you call the rendering methods.

For more information, see About Vertex Formats.

Strided Vertex Format

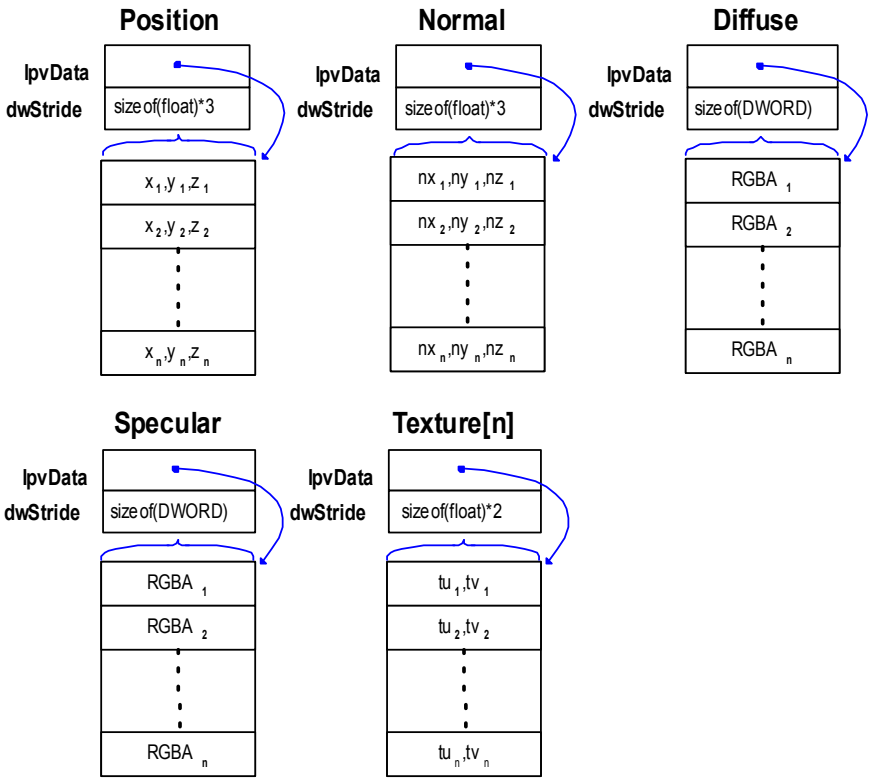
[This is preliminary documentation and subject to change.]

The strided vertex format contains fields to represent untransformed vertices, lit or unlit, for the **IDirect3DDevice3::DrawPrimitiveStrided** and **IDirect3DDevice3::DrawIndexedPrimitiveStrided** methods. You cannot use the strided vertex format for transformed vertices. Unlike a "normal" vertex, which is a structure that physically contains all the necessary vertex components, a "strided" vertex is a structure that contains pointers to the vertex components rather than the components themselves.

This indirection is accomplished through the **D3DDRAWPRIMITIVESTRIDEDDATA** structure that is accepted by the **DrawPrimitiveStrided** and **DrawIndexedPrimitiveStrided** methods. You describe strided vertices using the same combinations of flexible vertex format flags as a non-strided vertex. However, unlike non-strided vertices, the flags you use do not indicate the presence of a given field in memory (**D3DDRAWPRIMITIVESTRIDEDDATA** includes them all), they indicate which the structure members that your application uses. The restrictions for these flags are identical to non-strided vertices, for details, see Flexible Vertex Format Flags.

The **D3DDRAWPRIMITIVESTRIDEDDATA** structure contains 12 **D3DDP_PTRSTRIDE** structures, one structure each for the position, normal, diffuse color, specular color, and texture coordinate sets for the vertices. Each **D3DDP_PTRSTRIDE** structure contains a pointer to an array of data, and the stride of that array. These 12 structures provide the indirection that allows your application

to arrange vertex components however it needs. For instance, you might use distinct arrays for every vertex component, as shown in the following illustration.

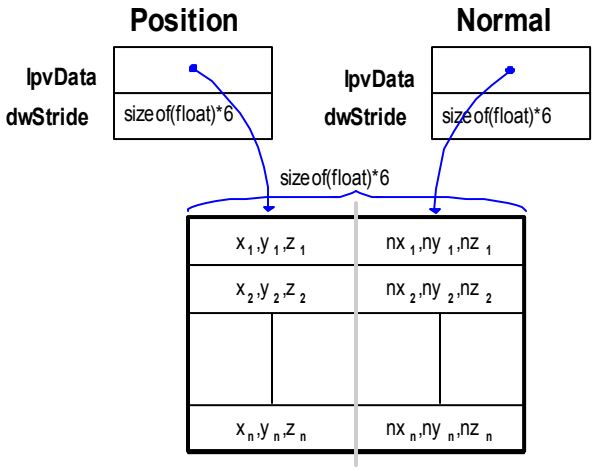


The **lpvData** member of the corresponding **D3DDP_PTRSTRIDE** structures contains the address of a buffer that contains an array of vertex components. Each element in each array represents a component for one vertex, which is made up of some number of floats (for position, normal, and texture coordinates) or an RGBA value (for diffuse and specular colors). There is one entry within each array for every vertex to be rendered. The **dwStride** member of **D3DDP_PTRSTRIDE** should be set to the memory stride, in bytes, from one entry in the array to the next. The following table describes the components and the corresponding strides for each.

Component	Stride
untransformed position	3 floats (x,y,z)
vertex normal	3 floats (nx,ny,nz)
diffuse color	1 DWORD (RGBA)
specular color	1 DWORD (RGBA)
texture coordinate set	2 floats (u,v)

Some developers might choose to include some or all vertex components in an interleaved array. The strided vertex format makes doing this very simple. If your

application interleaves the vertex position and normal components, for example, you could visualize the memory layout and corresponding settings in **D3DDP_PTRSTRIDE** structures as follows:



In this case, the **lpvData** members of the **D3DDP_PTRSTRIDE** structures point to two locations within the same buffer, and the strides are set to the combined width of the interleaved components. Of course, you aren't limited to any particular interleaving scheme. So long as you set the data pointers and their strides correctly, any interleaving scheme will work.

Textures

[This is preliminary documentation and subject to change.]

Textures are a powerful tool in the quest for realism in computer-generated 3-D images. Direct3D supports an extensive texturing feature set, providing developers with easy access to advanced texturing techniques. This section discusses the purposes and uses of textures in Direct3D. The information is presented in the following topics:

- Basic Texturing Concepts
- Texture Handles
- Texture Interfaces
- Texture Filtering
- Texture Wrapping
- Texture Blending
- Texture Compression
- Automatic Texture Management
- Hardware Considerations for Texturing

If you are familiar with what textures are and how they are used, you may want to skip the Basic Texturing Concepts section. It is important to note, however, that Direct3D now uses texture interfaces to access many of its most powerful texturing features. Therefore, you may want to look over the section entitled Texture Interfaces.

Basic Texturing Concepts

[This is preliminary documentation and subject to change.]

This section presents the most fundamental concepts required for an understanding of texturing in Direct3D. The information in this section is presented in the following topics:

- What Is a Texture?
- Texture Coordinates
- Texture Addressing Modes
- Texture Handles and Texture Interfaces
- Palettized Textures

What Is a Texture?

[This is preliminary documentation and subject to change.]

Early computer-generated 3-D images, although generally advanced for their time, tended to have a shiny plastic look. They lacked the types of markings that give 3-D objects realistic visual complexity such as scuffs, cracks, fingerprints, and smudges. In recent years, textures have gained popularity among developers as a tool for enhancing the realism of computer-generated 3-D images.

At its most basic, a texture is simply a bitmap of pixel colors. In this sense, the word texture has a specific definition when used in the context of computer graphics. In the normal semantics associated with the word texture, we refer both to the patterns of color on an object and its roughness or smoothness. Direct3D textures don't add "bumpiness" to an object. Rather, the textures, or patterns of colors, just give it the appearance of bumpiness.

Because Direct3D textures are simply bitmaps, any bitmap can be applied to a Direct3D primitive. For instance, applications can create and manipulate objects that appear to have a wood grain pattern in them. Grass, dirt, and rocks can be applied to a set of 3-D primitives that form a hill. The result is a very realistic-looking hillside. Texturing can also be used to create effects such as signs along a roadside, rock strata in a cliff, or the appearance of marble on a floor.

In addition, Direct3D supports more advanced texturing techniques such as texture blending (with or without transparency) and light mapping. Information on these techniques is presented in Texture Blending and Light Mapping With Textures.

If your application creates a HAL device, an MMX device, or an RGB device (see Direct3D Device Types), it can use 8-, 16-, 24-, or 32-bit textures. Legacy applications that use the monochromatic (or ramp) device must use 8-bit textures.

Texture Coordinates

[This is preliminary documentation and subject to change.]

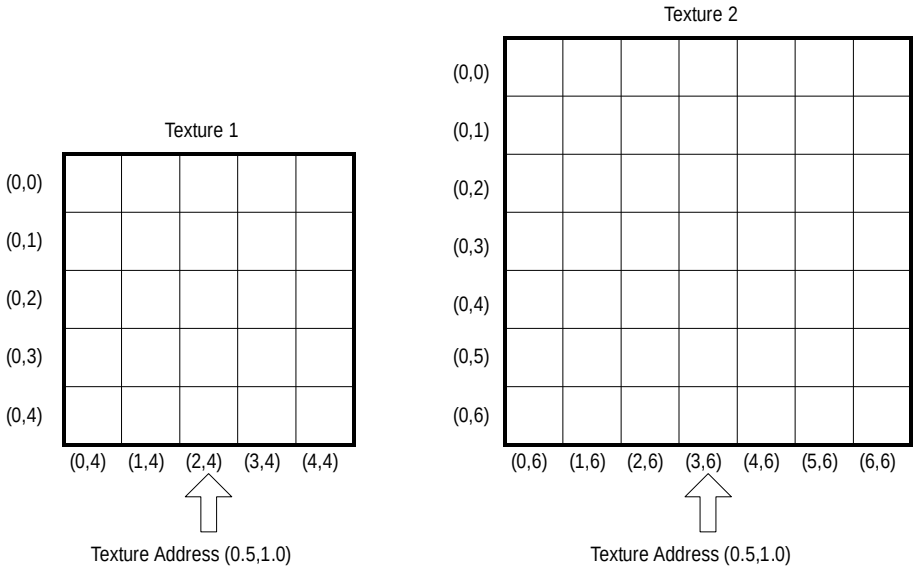
Textures, like most bitmaps, are a two dimensional array of color values. The individual color values are called texture elements, or texels. Each texel has a unique address in the texture. The address can be thought of as a column and row number, which are labeled U and V respectively.

Texture coordinates are in texture space. That is, they are relative to the location (0,0) in the texture. When a texture is applied to a primitive in 3-D space, its texel addresses must be mapped into object coordinates. They must then be translated into screen coordinates, or pixel locations.

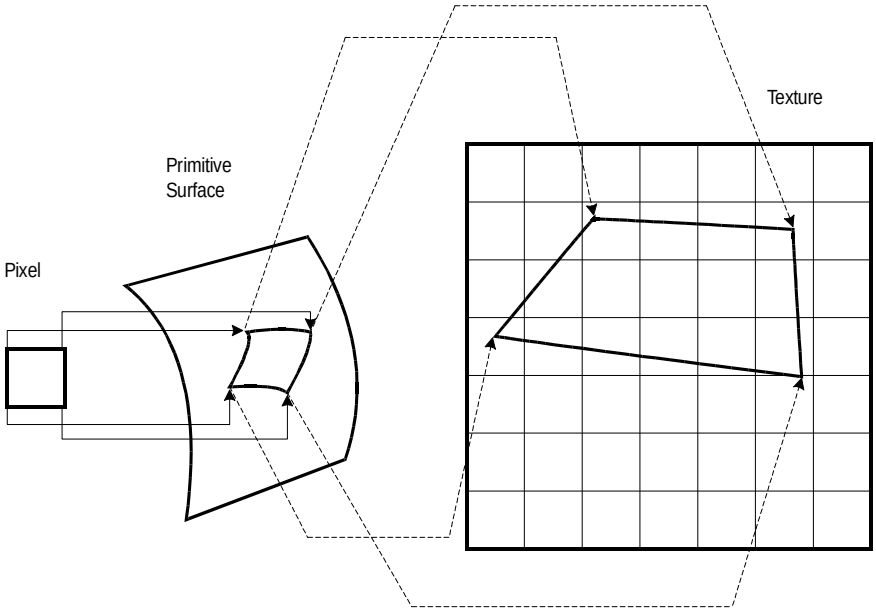
Direct3D maps texels in texture space directly to pixels in screen space, skipping the intermediate step for greater efficiency. This mapping process is actually an inverse mapping. That is, for each pixel in screen space, the corresponding texel position in texture space is calculated. The texture color at or around that point is sampled. The sampling process is called texture filtering. For more information, see Texture Filtering.

Each texel in a texture can be specified by its texel coordinate. However, in order to map texels onto primitives, Direct3D requires a uniform address range for all texels in all textures. Therefore, it uses a generic addressing scheme in which all texel addresses are in the range of 0.0 to 1.0 inclusive. Direct3D programs specify texture coordinates in terms of U,V values, much like 2-D Cartesian coordinates are specified in terms of x,y coordinates.

A result of this is that identical texture addresses can map to different texel coordinates in different textures. In the following illustration, the texture address being used is (0.5,1.0). However, because the textures are different sizes, the texture address maps to different texels. Texture 1, on the left, is 5x5. The texture address (0.5,1.0) maps to texel (2,4). Texture 2, on the right, is 7x7. The texture address (0.5,1.0) maps to texel (3,6).



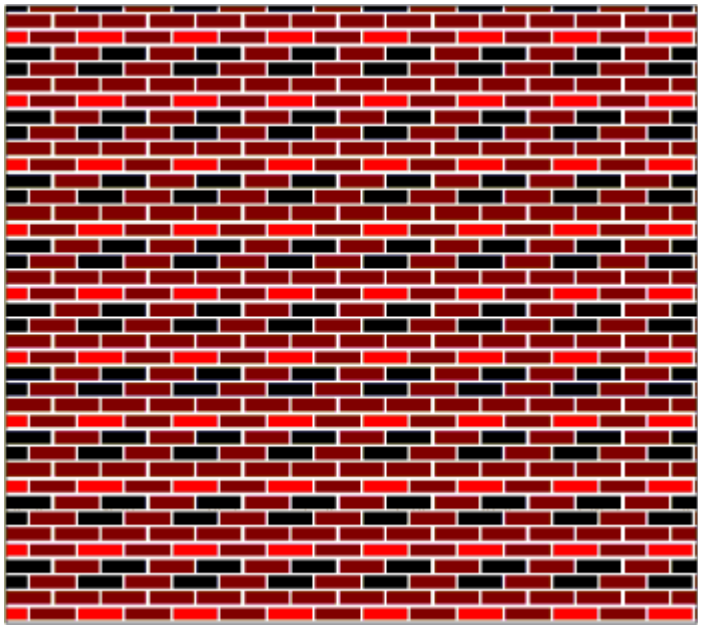
A simplified version of the texel mapping process is shown in the following diagram.



For this example, we are idealizing a pixel, shown at the left of the illustration, into a square of color. The addresses of the four corners of the pixel are mapped onto the 3-D primitive in object space. The shape of the pixel is often distorted because of the shape of the primitive in 3-D space and because of the viewing angle. The corners of the surface area on the primitive that correspond the corners of the pixel are then

mapped into texture space. The mapping process distorts the pixel's shape again, which is common. The final color value of the pixel is computed from the texels in the region to which the pixel maps. You determine the method that Direct3D uses to arrive at the pixel color when you set the texture filtering method. For more information, see Texture Filtering.

Your application can assign texture coordinates directly to vertices. For details, see **D3DVERTEX**. This capability gives you control over which portion of a texture is mapped onto a primitive. For instance, suppose you create a rectangular primitive that is exactly the same size as the texture in the following illustration. In this example, you want your application to map the whole texture onto the whole wall. The texture coordinates your application would assign to the vertices of the primitive are (0.0,0.0), (1.0,0.0), (1.0,1.0), and (0.0,1.0).



Let's say you decide to decrease the height of the wall by one-half. You can either distort the texture to fit onto the smaller wall, or you can assign texture coordinates that will cause Direct3D to use the bottom half of the texture.

If you decide to distort or scale the texture to fit the smaller wall, the texture filtering method that you use will influence the quality of the image. For more information, see Texture Filtering.

If, instead, you decide to assign texture coordinates to make Direct3D use the bottom half of the texture for the smaller wall, the texture coordinates your application would assign to the vertices of the primitive in this example are (0.0,0.0), (1.0,0.0), (1.0,0.5), and (0.0,0.5). Direct3D will apply the bottom half of the texture to the wall.

It is possible for texture coordinates of a vertex to be greater than 1.0. When you assign texture coordinates to a vertex that are not in the range of 0.0 to 1.0 inclusive,

you should also set the texture addressing mode. For further information, see Texture Addressing Modes.

Texture Addressing Modes

[This is preliminary documentation and subject to change.]

This section describes the purpose and use of Direct3D texture addressing modes. It is organized into the following topics:

- What Are Texture Addressing Modes?
- About the Wrap Texture Address Mode
- About the Mirror Texture Address Mode
- About the Clamp Texture Address Mode
- About the Border Color Texture Address Mode
- Setting and Retrieving Texture Addressing Modes
- Texture Addressing Modes and Texture Wrapping

What Are Texture Addressing Modes?

[This is preliminary documentation and subject to change.]

Your Direct3D application can assign texture coordinates to any vertex of any primitive. For details, see Texture Coordinates. Typically, the U and V texture coordinates that you assign to a vertex will be in the range of 0.0 to 1.0 inclusive. However, by assigning texture coordinates outside that range, you can create certain special texturing effects.

You control what Direct3D does with texture coordinates that are outside the [0.0, 1.0] range by setting the texture addressing mode. For instance, you can have your application set the texture addressing mode such that a texture is tiled across a primitive. The following topics contain additional details:

- About the Wrap Texture Address Mode
- About the Mirror Texture Address Mode
- About the Clamp Texture Address Mode
- About the Border Color Texture Address Mode

About the Wrap Texture Address Mode

[This is preliminary documentation and subject to change.]

The "wrap" texture address mode, identified by the D3DTADDRESS_WRAP member of the **D3DTEXTUREADDRESS** enumerated type, makes Direct3D repeat the texture on every integer junction. Suppose, for example, your program creates a square primitive and specifies texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0). Setting the texture addressing mode to D3DTADDRESS_WRAP will result

in the texture being applied three times in both the U and V directions. This is illustrated in the following figure.

The effects of this texture address mode are similar to, but distinct from, those of the "mirror" mode. For more information, see About the Mirror Texture Address Mode.

About the Mirror Texture Address Mode

[This is preliminary documentation and subject to change.]

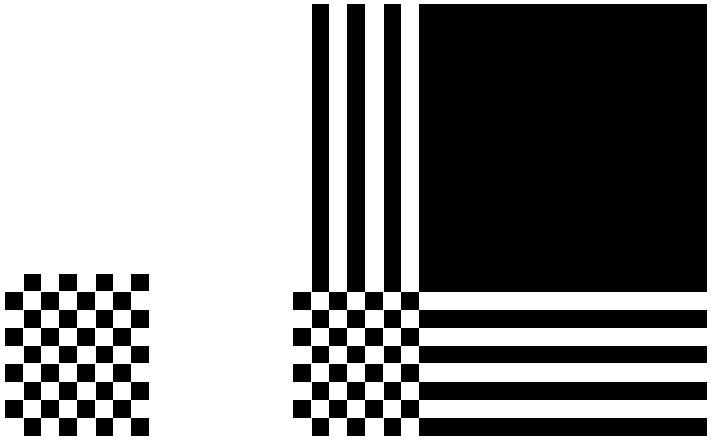
The "mirror" texture address mode, identified by the D3DTADDRESS_MIRROR member of the **D3DTEXTUREADDRESS** enumerated type, causes Direct3D will mirror the texture at every integer boundary. Suppose, for example, your program creates a square primitive and specifies texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0). Setting the texture addressing mode to D3DTADDRESS_MIRROR will result in the texture being applied three times in both the U and V directions. Every other row and column that it is applied to will be a mirror image of the preceding row or column. This is illustrated in the following figure.

The effects of this texture address mode are similar to, but distinct from, those of the "wrap" mode. For more information, see About the Wrap Texture Address Mode.

About the Clamp Texture Address Mode

[This is preliminary documentation and subject to change.]

The "clamp" texture address mode, identified by the D3DTADDRESS_CLAMP member of the **D3DTEXTUREADDRESS** enumerated type, causes Direct3D to clamp your texture coordinates to the [0.0, 1.0] range. That is, it will apply the texture once, then "smear" the color of edge pixels. For instance, suppose that your program creates a square primitive and assigns texture coordinates of (0.0,0.0), (0.0,3.0), (3.0,3.0), and (3.0,0.0) to the primitive's vertices. Setting the texture addressing mode to D3DTADDRESS_CLAMP will result in the texture being applied once. The pixel colors at the top of the columns and the end of the rows are extended to the top and right of the primitive respectively. This is illustrated in the following figure:



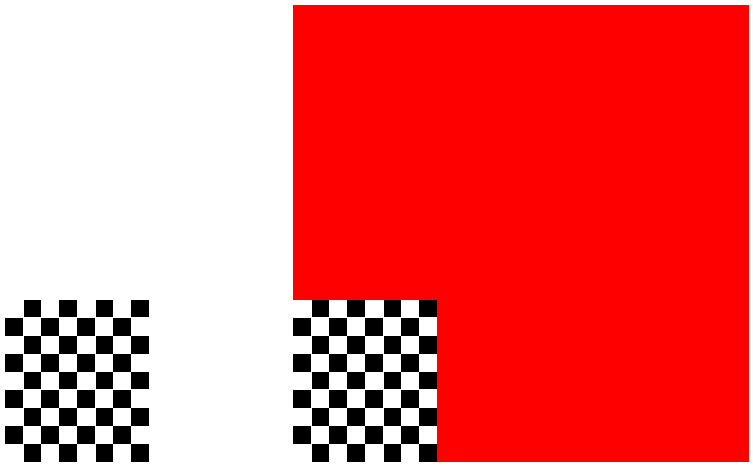
Texture Clamped texture applied to primitive

About the Border Color Texture Address Mode

[This is preliminary documentation and subject to change.]

The "border color" texture address mode, identified by the D3DTADDRESS_BORDER member of the **D3DTEXTUREADDRESS** enumerated type, causes Direct3D to use an arbitrary color, known as the border color, for any texture coordinates outside the range of 0.0 through 1.0, inclusive.

This is shown in the next illustration in which the application specified that the texture be applied to the primitive using a red border.



Texture Texture with red border applied to primitive

How your application sets the border color depends on what version of the Direct3D device interface it uses. If your application uses the **IDirect3DDevice3** interface, set the border color by calling **IDirect3DDevice3::SetTextureStageState**. Set the first parameter for the call to the desired texture stage identifier, the second parameter to

the D3DTSS_BORDERCOLOR stage state value, and the third parameter to the new RGBA border color.

If your application stills uses the legacy **IDirect3DDevice2** interface, you can set the border color by calling **IDirect3DDevice2::SetRenderState** method, specifying the D3DRENDERSTATE_BORDERCOLOR render state value and the new RGBA border color as parameters.

Note

The **IDirect3DDevice3::SetRenderState** method (as opposed to the **IDirect3DDevice2** version) still recognizes the D3DRENDERSTATE_BORDERCOLOR render state, even though it has been superseded. Instead of failing this legacy render state, the **IDirect3DDevice3** implementation maps the effect of this render state to the first texture stage (stage 0). Applications should not mix the legacy render states with their corresponding texture stage states, as unpredictable results can occur.

Setting and Retrieving Texture Addressing Modes

[This is preliminary documentation and subject to change.]

How your application sets and retrieves texture addressing modes depends largely on which version of the Direct3D device interface it uses. When performing multiple texture blending by way of the **IDirect3DDevice3** interface, you can set texture addressing modes for individual texture stages by calling the **IDirect3DDevice3::SetTextureStageState** method. Specify the desired texture stage identifier in the first parameter. Set the second parameter to D3DTSS_ADDRESS to change both the U and V texture addressing modes simultaneously, or use the D3DTSS_ADDRESSU or D3DTSS_ADDRESSV values to update the U or V addressing modes individually. The third parameter you pass to **SetTextureStageState** determines which mode is being set; this can be any one of the members of the **D3DTEXTUREADDRESS** enumerated type. To retrieve the current texture address mode for a given texture stage, call **IDirect3DDevice3::GetTextureStageState**, using the D3DTSS_ADDRESS, D3DTSS_ADDRESSU, or D3DTSS_ADDRESSV members of the **D3DTEXTURESTAGESTATETYPE** enumeration to identify about which address mode you want information.

If your application uses the **IDirect3DDevice2** interface, you set texture addressing modes by calling the **IDirect3DDevice2::SetRenderState** method, using the D3DRENDERSTATE_TEXTUREADDRESS render state to simultaneously set U and V texture addressing. You can set U and V addressing individually with the D3DRENDERSTATE_TEXTUREADDRESSU or D3DRENDERSTATE_TEXTUREADDRESSV render states. Like their newer **SetTextureStageState** counterparts, these render states also use the values from **D3DTEXTUREADDRESS** enumerated type.

Note

The **IDirect3DDevice3::SetRenderState** method (as opposed to the **IDirect3DDevice2** version) still recognizes the D3DRENDERSTATE_TEXTUREADDRESS, D3DRENDERSTATE_TEXTUREADDRESSU, D3DRENDERSTATE_TEXTUREADDRESSV render states, even though they have been superseded. Instead of failing these legacy render states, the **IDirect3DDevice3** implementation maps their effects to the first texture stage (stage 0). Applications should not mix the legacy render states with their corresponding texture stage states, as unpredictable results can occur.

Texture Addressing Modes and Texture Wrapping

[This is preliminary documentation and subject to change.]

Direct3D enables applications to perform texture wrapping. It is important to note that setting the texture addressing mode to D3DTADDRESS_WRAP is not the same as performing texture wrapping. Setting the texture addressing mode to D3DTADDRESS_WRAP results in multiple copies of the source texture being applied to the current primitive, and enabling texture wrapping changes how the system rasterizes textured polygons. For details, see Texture Wrapping.

Enabling texture wrapping effectively makes texture coordinates outside the [0.0, 1.0] range invalid, and the behavior for rasterizing such "delinquent" texture coordinates is undefined in this case. When texture wrapping is enabled, texture addressing modes are not used. Take care that your application does not specify texture coordinates lower than 0.0 or higher than 1.0 when texture wrapping is enabled.

Texture Handles and Texture Interfaces

[This is preliminary documentation and subject to change.]

Direct3D provides two methods of manipulating and controlling textures, texture handles and texture interfaces. Texture handles are obsolete. They were used in applications that utilized the **IDirect3D** and **IDirect3D2** interfaces. For details, see Texture Handles.

With the introduction of the **IDirect3D3** interface, you now create and use textures through texture interface pointers. Obtain a texture interface pointer by querying the DirectDrawSurface object for the **IDirect3DTexture2** interface by calling the surface's **IUnknown::QueryInterface** method (use the IID_IDirect3DTexture2 reference identifier).

With the advent of texture interfaces in Direct3D, powerful new texturing features are also realized. Texture interfaces support the blending of up to eight textures onto a primitive at once. New texture blending operations have been also added. For details, see Texture Interfaces.

Palettized Textures

[This is preliminary documentation and subject to change.]

Direct3D devices can support texturing from texture surfaces that use attached palettes. These types of textures are sometimes called "palettized textures." A palettized texture is a DirectDrawSurface object, created with the DDSCAPS_TEXTURE capability, that uses one of the DDPF_PALETTEINDEXED*n* pixel formats (where *n* is 1, 2, 4, or 8). These capabilities are included in the **DDSURFACEDESC2** structure that you use to create a texture (and the **DDPIXELFORMAT** structure that the description contains). Like all palettized surfaces, instead of a color, each pixel is an index into a table of values held within an attached DirectDrawPalette object. For more information about creating and using palettized surfaces, see Surfaces and Palettes in the DirectDraw documentation. As you should always do when checking texture-related device capabilities, be sure to verify the supported texture pixel formats by calling the **IDirect3DDevice3::EnumTextureFormats** method.

To prepare a palettized texture

1. Check DirectDraw and Direct3D capabilities as described in this section.
2. Create a surface of the desired dimensions that includes the DDSCAPS_TEXTURE capability and uses one of the DDPF_PALETTEINDEXED*n* pixel formats.
3. Create and initialize a DirectDrawPalette object. For more information, see Palettes in the DirectDraw documentation. (To use an alpha-only palettized texture, include the DDPCAPS_ALPHA capability when you create the palette. See the reference for **IDirectDraw4::CreatePalette** for more information.)
4. Attach the palette to the surface by calling the **IDirectDrawSurface4::SetPalette** method for the texture surface.

Note

If you create a palettized texture surface, but neglect to attach a palette, your application will cause an access violation within Direct3D Immediate Mode during rendering.

The palettes you use with palettized textures need not be limited to color data. In some cases, devices support texture palettes that also contain alpha information. If so, DirectDraw will expose the DDPCAPS_ALPHA palette capability flag when you call the **IDirectDraw4::GetCaps** method—the flag is found in the **dwPalCaps** member of the associated **DDCAPS** structure. If the rendering device can perform texturing from alpha-capable palettized textures, it will expose the D3DPTEXTURECAPS_ALPHAPALETTE capability when you call **IDirect3DDevice3::GetCaps**. (The D3DPTEXTURECAPS_ALPHAPALETTE flag can be found in the two **D3DPRIMCAPS** structures contained by the **D3DDEVICEDESC** structure you pass with the call.)

Texture Handles

[This is preliminary documentation and subject to change.]

Texture handles are primarily provided for backward compatibility. New applications should use texture interface pointers. See Texture Interfaces.

The **IDirect3D** and **IDirect3D2** interfaces are used when programming with Direct3D texture handles. A Direct3D texture is a DirectDraw surface. You can use a DirectDraw surface as a texture map by calling the **IDirectDrawSurface4::QueryInterface** method to retrieve an **IDirect3DTexture2** interface. Use the **IDirect3DTexture2** interface to load textures, retrieve handles, and track changes to palettes.

A texture map created with the **IDirect3DTexture2** interface must be associated with a 3-D device. A texture handle identifies the coupling of a texture map and a device. A given texture can be associated with more than one device. When your application calls the **IDirect3DTexture2::GetHandle** method to associate a texture with a device, Direct3D validates it to ensure that the device supports the specified type of texture format and dimensions. The **GetHandle** method returns the texture handle if this validation succeeds. Your application can then use texture handles as parameters to render states. For details, see Render States.

Applications can use handles obtained using **IDirect3DTexture** or **IDirect3DTexture2** for a given device interface interchangeably.

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

This section presents information on creating and rendering with texture handles in the topics listed hereafter:

- Creating a Texture Handle
- Rendering with Texture Handles

Creating a Texture Handle

[This is preliminary documentation and subject to change.]

The following example demonstrates how to create an **IDirect3DTexture2** interface. It also illustrates the process of obtaining a texture handle by calling the **IDirect3DTexture2::GetHandle** method. It then loads the texture using the **IDirect3DTexture2::Load** method. Note that the DirectDraw surface you query must have the DDSCAPS_TEXTURE capability to support a Direct3D texture. See Creating Surfaces and **DDSCAPS**.

```
// This code fragment assumes that lpDDS is a valid pointer to
// a DirectDraw surface, and that lpD3DDevice is a valid pointer to
// an IDirect3DDevice3 interface.

LPDIRECT3DTEXTURE2 lpD3DTexture2;
D3DTEXTUREHANDLE d3dhTextureHandle;

// Get the texture interface pointer.
lpDDS->QueryInterface( IID_IDirect3DTexture2,
```

```
                &lpD3DTexture2);

// Associate the texture with a device.
lpD3DTexture2->GetHandle( lpD3DDevice,
                d3dhTextureHandle);

// Load the texture.
lpD3DTexture2->Load(lpD3DTexture2);
```

Rendering with Texture Handles

[This is preliminary documentation and subject to change.]

After your program creates a texture handle and loads a bitmap onto the texture's surface, it may use the texture for rendering. Texture handles can be used to set the texture-related rendering states whether your application utilizes the DrawPrimitive methods or execute buffers. If it uses the DrawPrimitive methods, all rendering states are set by invoking the **IDirect3DDevice3::SetRenderState** method. Pass the texture-related rendering state as the first parameter. The second parameter must be the texture's handle.

When an application sets a texture as the current texture, Direct3D applies it to all primitives that it renders with the DrawPrimitive methods. For further information, see Current Texture.

Your application controls the mapping of texture coordinates to screen coordinates by setting the texture-addressing rendering state. See Texture Addressing State.

On some 3-D hardware, applications can also enable or disable perspective correction. Many 3-D cards apply texture perspective correction unconditionally. Direct3D perspective correction is enabled by default. See Texture Perspective State.

Direct3D sports a powerful set of texture filtering capabilities. For further information, see Texture Filtering and Texture Filtering State.

When it applies a texture to a primitive's surface, your application can blend the texture's texel colors with the current color of a primitive. It can only blend one texture at a time if it uses texture handles. See Texture Blending State.

Applications that render with execute buffers use texture handles. Invoke the **IDirect3DDevice3::SetRenderState** method and pass the D3DRENDERSTATE_TEXTUREHANDLE render state (part of the **D3DRENDERSTATETYPE** enumerated type) as the value of the first parameter. Set the texture handle as the second parameter.

Texture Interfaces

[This is preliminary documentation and subject to change.]

Applications that create an **IDirect3D3** interface can take advantage of the new texturing features in Direct3D. Using **IDirect3DTexture2** interface pointers, rather than texture handles, Direct3D can now perform multiple texturing if the user's hardware supports it. For more information, see Multiple Texture Blending.

This section presents information on creating and rendering with texture interface pointers in the following topics:

- Obtaining a Texture Interface Pointer
- Rendering with Texture Interface Pointers

Obtaining a Texture Interface Pointer

[This is preliminary documentation and subject to change.]

Textures under the **IDirect3D3** interface are DirectDraw surfaces, just as textures are when programming with texture handles. Therefore, the first step in obtaining a texture interface pointer is to create a DirectDraw surface with the DDSCAPS_TEXTURE capability set. See Creating Surfaces , **DDSCAPS**, and **DDSCAPS2**.

Once the surface is created, your application can retrieve a pointer to its texture interface. To do so, simply query the surface for its **IDirect3DTexture2** interface.

The following code fragment illustrates the process of creating a texture from an existing DirectDrawSurface object's interface. It also demonstrates how to load a texture.

```
// This code fragment assumes that lpDDS is a valid pointer to
// a DirectDraw surface that was created with the DDSCAPS_TEXTURE
// surface capability, and that lpD3D3 is a valid pointer to
// an IDirect3D3 interface.

LPDIRECT3DTEXTURE2 lpD3DTexture2;

// Get the texture interface.
lpDDS->QueryInterface(IID_IDirect3DTexture2, (void**)&lpD3DTexture2);

// Load the texture.
lpD3DTexture2->Load(lpD3DTexture2);
```

Note

Loading a texture with the **IDirect3DTexture2::Load** method allocates the memory for the texture. Once this is done, a bitmap may be loaded from a resource or a file onto the surface associated with the texture by using DirectDraw blitting methods. (In the event that blitting isn't supported, you can access the surface memory directly by using the DirectDraw **IDirectDrawSurface4::Lock** and **IDirectDrawSurface4::Unlock** methods.)

Rendering with Texture Interface Pointers

[This is preliminary documentation and subject to change.]

As part of the multitexturing functionality of that was introduced in the **IDirect3DDevice3** interface, Direct3D supports texture stages. Each texture stage contains a texture and operations that can be performed on the texture. The textures in the texture stages form the set of current textures. For more information, see Multiple Texture Blending. The state of each texture is encapsulated in its texture stage. Therefore, the state of each texture must be set with the **IDirect3DDevice3::SetTextureStageState** method. Pass the stage number (0-7) as the value of the first parameter. Set the value of the second parameter to a member of the member of the **D3DTEXTURESTAGESTATETYPE** enumerated type. The final parameter is the state value for the particular texture state.

Using texture interface pointers, your application can render a blend of up to eight textures. Set the current textures by invoking the **IDirect3DDevice3::SetTexture** method. Direct3D will blend all current textures onto the primitives that it renders.

Your application can set the texture wrapping state for the current textures by calling the **IDirect3DDevice3::SetRenderState** method. Pass a value from D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7 as the value of the first parameter, and use a combination of the D3DWRAP_U or D3DWRAP_V flags to enable wrapping in the u or v directions.

Your application can also set the texture perspective and texture filtering states. See Texture Perspective State, Texture Filtering, and Texture Filtering State.

Texture Filtering

[This is preliminary documentation and subject to change.]

When Direct3D renders a primitive, it maps the 3-D primitive onto a 2-D screen. If the primitive has a texture, Direct3D must use that texture to produce a color for each pixel in the primitive's 2-D rendered image. For every pixel in the primitive's on-screen image, it must obtain a color value from the texture. This process is called texture filtering.

When a texture filter operation is performed, the texture being used is typically also being magnified or minified. In other words, it is being mapped onto a primitive image that is larger or smaller than itself. Magnification of a texture can result in many pixels being mapped to one texel. The result can be a chunky appearance. Minification of a texture often means that a single pixel is mapped to many texels. The resulting image can be blurry or aliased. To resolve these problems, some blending of the texel colors must be performed to arrive at a color for the pixel.

Direct3D simplifies the complex process of texture filtering. It provides developers with three types of texture filtering — linear filtering, anisotropic filtering, and mipmap filtering. If you select no texture filtering, Direct3D utilizes a technique called nearest point sampling.

Each type of texture filtering has advantages and disadvantages. For instance, linear texture filtering can produce jagged edges or a chunky appearance in the final image. However, it is a computationally low-overhead method of texture filtering. On the other hand, filtering with mipmaps usually produces the best results, especially when combined with anisotropic filtering. However it requires the most memory of the techniques that Direct3D supports.

If your application uses texture handles, it should set the current texture filtering method by invoking the **IDirect3DDevice3::SetRenderState** method. The first parameter must be either D3DRENDERSTATE_TEXTUREMAG or D3DRENDERSTATE_TEXTUREMIN. It must pass a member of the **D3DTEXTUREFILTER** enumerated type as the value of the second parameter. See Texture Filtering State.

Applications that use texture interface pointers should set the current texture filtering method by calling the **IDirect3DDevice3::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass either D3DTSS_MAGFILTER, D3DTSS_MINFILTER, or D3DTSS_MIPFILTER as the value of the second parameter. Set the third parameter to a member of the **D3DTEXTUREMAGFILTER**, **D3DTEXTUREMINFILTER**, or **D3DTEXTUREMIPFILTER** enumerated types respectively.

This section presents the texture filtering methods that Direct3D supports. It is organized into the following topics:

- Nearest Point Sampling
- Linear Texture Filtering
- Anisotropic Texture Filtering
- Texture Filtering With Mipmaps

Nearest Point Sampling

[This is preliminary documentation and subject to change.]

Applications are not required to use texture filtering. Direct3D can be set so that it computes the texel address, which often does not evaluate to integers, and simply copies the color of the texel with the closest integer address. This process is called nearest point sampling. This can be a fast and efficient way to process textures if the size of the texture is similar to the size of the primitive's image on the screen. If not, the texture will need to be magnified or minified. The result can be a chunky, aliased, or blurred image.

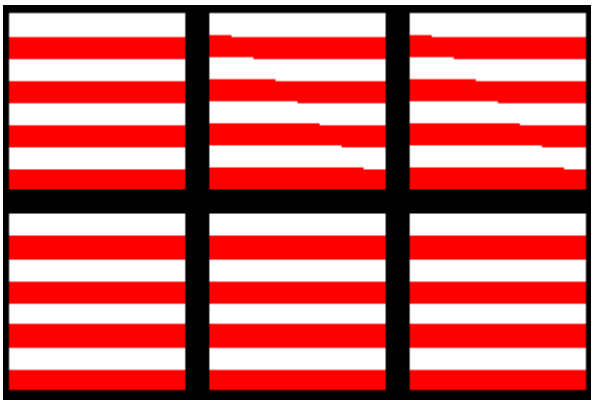
Your application can select nearest point sampling by calling the **IDirect3DDevice3::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass **D3DTEXTUREMAGFILTER** as the value of the second parameter if you are setting the magnification filter. Pass **D3DTEXTUREMIPFILTER** as the value of the second parameter if you are setting

the minification filter. Pass **D3DTEXTUREMIPFILTER** as the value of the second parameter if you are setting the mipmapping filter. Set the third parameter to D3DTFG_POINT if you are setting the magnification filter, D3DTFN_POINT if you are setting the minification filter, or D3DTFP_POINT if you are setting the mipmap filter. For more information, see Texture Filtering State.

You should use nearest point sampling carefully, as it can sometimes cause graphic artifacts when the texture is sampled at the boundary between two texels. This boundary is the position along the texture (u or v) at which the sampled texel transitions from one texel to the next. When point sampling is used, the system chooses one sample texel or the other, and the result can change abruptly from one texel to the next texel as the boundary is crossed. This effect can appear as undesired graphic artifacts in the displayed texture. (When linear filtering is used, the resulting texel is computed from both adjacent texels and smoothly blends between them as the texture index moves through the boundary.)

This effect can be seen when mapping a very small texture onto a very large polygon: an operation often called "magnification." For example, when using a texture that looks like a checkerboard, nearest point sampling results in a larger checkerboard that shows distinct edges. By contrast, linear texture filtering results in an image where the checkerboard colors vary smoothly across the polygon.

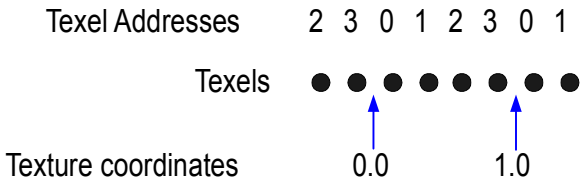
In most cases, applications can receive the best results by avoiding nearest point sample wherever possible. The majority of hardware today is optimized for linear filtering, so your application should not suffer degraded performance. If the effect you desire absolutely requires the use of the nearest point sampling—such as when using textures to display readable text characters—then your application should be extremely careful to avoid sampling at the texel boundaries, which might result in undesired effects. The following screen capture shows what these artifacts can look like:



Notice that the two squares in the top-right of the group appear different that their neighbors. To avoid graphic artifacts like these, you should start by understanding Direct3D texture sampling rules for nearest-point filtering. Direct3D maps a floating-point texture coordinate ranging from [0.0, 1.0] (0.0 to 1.0, inclusive) to an integer texel space value ranging from [-0.5, *n* - 0.5], where *n* is the number of texels in a

given dimension on the texture. The resulting texture index is rounded to the nearest integer. This mapping can introduce sampling inaccuracies at texel boundaries.

For a simple example, imagine an application that renders polygons with the D3DTADDRESS_WRAP texture addressing mode. Using the mapping employed by Direct3D, the *u* texture index maps as follows for a texture with a width of 4 texels:



Notice that the texture coordinates — 0.0 and 1.0 — for this illustration are exactly at the boundary between texels. Using the method by which Direct3D maps values, the texture coordinates range from [-0.5, 4 - 0.5], where 4 is the width of the texture. For this case, the sampled texel will be the 0^{th} texel for a texture index of 1.0. However, if the texture coordinate was only slightly less than 1.0, the sampled texel would be the n^{th} texel instead of the 0^{th} texel.

The implication of this is that magnifying a small texture using texture coordinates of exactly 0.0 and 1.0 with nearest-point filtering on a screen-space aligned triangle will result in pixels for which the texture map is sampled at the boundary between texels. Any inaccuracies in the computation of texture coordinates, however small, will result in artifacts along the areas in the rendered image which correspond to the texel edges of the texture map.

Performing this mapping of floating point texture coordinates to integer texels with perfect accuracy is difficult, computationally expensive, and generally not necessary. Most hardware implementations use an iterative approach for computing texture coordinates at each pixel location within a triangle. Iterative approaches tend to hide these inaccuracies somewhat because the errors are accumulated evenly during iteration.

The Direct3D reference rasterizer uses a direct-evaluation approach for computing texture indices at each pixel location. Direct evaluation differs from the iterative approach in that any inaccuracy in the operation exhibits a more random error distribution. The result of this is that the sampling errors occurring at the boundaries can be more noticeable since the reference rasterizer does not perform this operation with perfect accuracy (which is, again, difficult and very expensive with floating point texture coordinates).

The best approach is to use nearest-point filtering only when necessary. When you must use it, it is recommended that you offset texture coordinates slightly from the boundary positions to avoid artifacts.

Linear Texture Filtering

[This is preliminary documentation and subject to change.]

Direct3D uses a form of linear texture filtering called bilinear filtering. Like nearest point sampling, bilinear texture filtering first computes a texel address, which is usually not an integer address. Just as in nearest point sampling, if then finds the texel whose integer address is closest the computed address. In addition, the Direct3D rendering module will compute a weighted average of the texels that are immediately above, below, to the left of, and to the right of the nearest sample point.

Select bilinear texture filtering by invoking the **IDirect3DDevice3::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass **D3DTEXTUREMAGFILTER** as the value of the second parameter if you are setting the magnification filter. Pass **D3DTEXTUREMIPFILTER** as the value of the second parameter if you are setting the minification filter. Pass **D3DTEXTUREMIPFILTER** as the value of the second parameter if you are setting the mipmapping filter. Set the third parameter to D3DTFG_LINEAR if you are setting the magnification filter, D3DTFN_LINEAR if you are setting the minification filter, or D3DTFP_LINEAR if you are setting the mipmap filter. For more information, see Texture Filtering State.

Anisotropic Texture Filtering

[This is preliminary documentation and subject to change.]

Anisotropy is the distortion visible in the texels of a 3-D object whose surface is oriented at an angle with respect to the plane of the screen. When a pixel from an anisotropic primitive is mapped into texels, its shape is distorted. Direct3D measures the anisotropy of a pixel as the elongation (length divided by width) of a screen pixel that is inverse-mapped into texture space.

Anisotropic texture filtering can be used in conjunction with linear texture filtering or mipmap texture filtering to improve rendering results. Your application enables anisotropic texture filtering by calling the **IDirect3DDevice3::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are selecting a texture filtering method. Pass **D3DTEXTUREMAGFILTER** as the value of the second parameter if you are setting the magnification filter. Pass **D3DTEXTUREMINFILTER** as the value of the second parameter if you are setting the minification filter. Set the third parameter to D3DTFG_ANISOTROPIC if you are setting the magnification filter, or D3DTFN_ANISOTROPIC if you are setting the minification filter. For more information, see Texture Filtering State.

Your program must also set the degree of anisotropy to a value greater than zero and less than one. Do this by calling the **IDirect3DDevice3::SetTextureStageState** method. Set the value of the first parameter to the integer index number (0-7) of the texture for which you are setting the degree of isotropy. Pass D3DTSS_MAXANISOTROPY as the value of the second parameter. The final parameter should be the degree of isotropy.

Disable isotropic filtering by setting the degree of isotropy to one (any value larger than one enables it). Check the D3DPRASTERCAPS_ANISOTROPY flag in the

D3DPRIMCAPS structure to determine the possible range of values for the degree of anisotropy.

Texture Filtering With Mipmaps

[This is preliminary documentation and subject to change.]

Mipmap textures are used in 3-D scenes to decrease the time required for rendering a scene. They also improve the scene's realism. However, they often require large amounts of memory.

This section presents the fundamentals of using mipmap textures in 3-D scenes in the following topics:

- What Is a Mipmap?
- Creating a Set of Mipmaps
- Selecting and Displaying a Mipmap

What Is a Mipmap?

[This is preliminary documentation and subject to change.]

A mipmap is a sequence of textures, each of which is a progressively lower resolution representation of the same image. The height and width of each image, or level, in the mipmap is a power of two smaller than the previous level. Mipmaps do not have to be square.

A high-resolution mipmap image is used for objects that are close to the viewer. Lower-resolution images are used as the object moves farther away. Mipmapping improves the quality of rendered textures at the expense of using more memory.

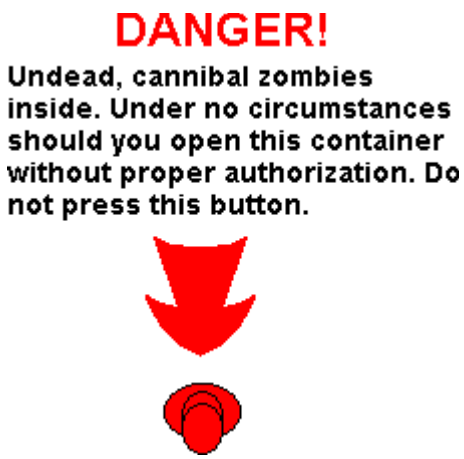
Direct3D represents mipmaps as a chain of attached surfaces. The highest resolution texture is at the head of the chain and has, as an attachment, the next level of the mipmap. That level has, in turn, an attachment that is the next level in the mipmap, and so on down to the lowest resolution level of the mipmap.

The following set of illustrations shows an example. The set of bitmap textures represents a sign on the side of a container in a 3-D, first-person game. When created as a mipmap, the highest-resolution texture is first in the set. Each succeeding texture in the mipmap set is a power of 2 smaller in height and width. In this case, the maximum-resolution mipmap is 256 pixels by 256 pixels. The next, texture is 128x128. The last texture in the chain is 64x64.

This sign would have a maximum distance from which it is visible. If the player begins far away from the sign, the game would display the smallest texture in the mipmap chain, which in this case the 64x64 texture.

As the player moves the point of view closer to the sign, progressively higher-resolution textures in the mipmap chain are used.

The highest-resolution texture is used when the user's point of view is at the minimum allowable distance from the sign.



This is a computationally lower-overhead way of simulating perspective effects for textures. Rather than render a single texture to many resolutions, it is faster to use multiple textures at varying resolutions.

Direct3D is able to assess which texture in a mipmap set is the closest resolution to the desired output and map pixels into its texel space. If the resolution of the final image is between the resolutions of the textures in the mipmap set, Direct3D can examine texels in both of the mipmaps and blend their color values together.

If you want your application to use mipmaps, it must build a set of mipmaps. For details, see Creating a Set of Mipmaps. If your program uses texture handles, it must then select the mipmap set as the current texture. For more information, see Current Texture. If it uses texture interface pointers, it must select the mipmap set as the first texture in the set of current textures. For more information, see Multiple Texture Blending.

Next, your program must set the filtering method that Direct3D uses to sample texels. The fastest method of mipmap filtering is to have Direct3D select the nearest texel. Use the D3DTFP_POINT enumerated value to select this. Direct3D can produce better filtering results if your application uses the D3DTFP_LINEAR enumerated

value. This will select the nearest mipmap, then compute a weighted average of the texels surrounding the location in the texture to which the current pixel maps.

Creating a Set of Mipmaps

[This is preliminary documentation and subject to change.]

To create a surface representing a single level of a mipmap, specify the DDSCAPS_MIPMAP and DDSCAPS_COMPLEX flags in the **DDSURFACEDESC** structure passed to the **IDirectDraw4::CreateSurface** method. Because all mipmaps are also textures, the DDSCAPS_TEXTURE flag must also be specified.

It is possible to create each level manually and build the chain by using the **IDirectDrawSurface4::AddAttachedSurface** method. However, this is not recommended. Many 3-D hardware vendors optimize their drivers for the **IDirectDraw4::CreateSurface** method. Applications that build mipmap chains with calls to **IDirectDrawSurface4::AddAttachedSurface** may find that mipmapping is not as fast.

The following example demonstrates how your application can use the **IDirectDraw4::CreateSurface** method to build a chain of five mipmap levels of sizes 256×256, 128×128, 64×64, 32×32, and 16×16:

```
// This code fragment assumes that the variable lpDD is a
// valid pointer to a DirectDraw interface.

DDSURFACEDESC       ddsd;
LPDIRECTDRAWSURFACE4 lpDDMipMap;
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_MIPMAPCOUNT;
ddsd.dwMipMapCount = 5;
ddsd.ddsCaps.dwCaps = DDSCAPS_TEXTURE |
   DDSCAPS_MIPMAP | DDSCAPS_COMPLEX;
ddsd.dwWidth = 256UL;
ddsd.dwHeight = 256UL;

ddres = lpDD->CreateSurface(&ddsd, &lpDDMipMap);
if (FAILED(ddres))
.
.
.
```

You can omit the number of mipmap levels, in which case the **IDirectDraw4::CreateSurface** method will create a chain of surfaces, each a power of two smaller than the previous one, down to the smallest possible size. It is also possible to omit the width and height, in which case **IDirectDraw4::CreateSurface** will create the number of levels you specify, with a minimum level size of 1×1.

Note

Each surface in a mipmap chain has dimensions that are one-half that of the previous surface in the chain. If the top-level mipmap has dimensions of 256×128, the dimensions of the second-level mipmap are 128×64, the third-level is 64×32, and so on down to 2×1. If you explicitly specify dimensions in the **dwWidth** and **dwHeight** members, you should be aware of some restrictions. Namely, you cannot request a number of mipmap levels in **dwMipMapCount** that would cause either the width or height of any mipmap in the chain to be smaller than 1. Take the very simple case of a 4×2 top-level mipmap surface: the maximum value allowed for **dwMipMapCount** here is 2: the top-level dimensions are 4×2, and the dimensions for the second level 2×1. A value larger than 2 in **dwMipMapCount** would result in a fractional value in the height of the second-level mipmap, and is therefore disallowed.

After your application creates the mipmap surfaces, it needs to associate the surface with a texture. If you are using texture handles, use the procedures outlined in Creating a Texture Handle. If you are using texture interface pointers, see Obtaining a Texture Interface Pointer.

Selecting and Displaying a Mipmap

[This is preliminary documentation and subject to change.]

If your program uses texture handles, it must assign the handle of the mipmap texture as the current texture. For details, see Current Texture.

If your application uses texture interface pointers, it must set the mipmap texture set as the first texture in the list of current textures. For more information, see Multiple Texture Blending.

After your application selects the mipmap texture set, it must assign values from the **D3DTEXTUREFILTER** enumerated type to the D3DRENDERSTATE_TEXTUREMAG and D3DRENDERSTATE_TEXTUREMIN render states. Direct3D will then automatically perform mipmap texture filtering.

Your application can also manually traverse a chain of mipmap surfaces by using the **IDirectDrawSurface4::GetAttachedSurface** method and specifying the DDSCAPS_MIPMAP and DDSCAPS_TEXTURE flags in the **DDSCAPS** structure. The following example traverses a mipmap chain from highest to lowest resolutions:

```
LPDIRECTDRAWSURFACE lpDDLevel, lpDDNextLevel;
DDSCAPS ddsCaps;
HRESULT ddres;

lpDDLevel = lpDDMipMap;
lpDDLevel->AddRef();
ddsCaps.dwCaps = DDSCAPS_TEXTURE | DDSCAPS_MIPMAP;
ddres = DD_OK;
while (ddres == DD_OK)
{
```

```
    // Process this level.
    .
    .
    .
    ddres = lpDDLevel->GetAttachedSurface(
       &ddsCaps, &lpDDNextLevel);
    lpDDLevel->Release();
    lpDDLevel = lpDDNextLevel;
}
if ((ddres != DD_OK) && (ddres != DDERR_NOTFOUND))
{
    // Code to handle the error goes here
}
.
.
.
```

Applications need to manually traverse a mipmap chain to load bitmap data into each surface in the chain. This is typically the only reason to traverse the chain.

Direct3D explicitly stores the number of levels in a mipmap chain. When an application obtains the surface description of a mipmap (by calling the **IDirectDrawSurface4::Lock** or **IDirectDrawSurface4::GetSurfaceDesc** method), the **dwMipMapCount** member of the **DDSURFACEDESC** structure contains the number of levels in the mipmap, including the top level. For levels other than the top level in the mipmap, the **dwMipMapCount** member specifies the number of levels from that mipmap to the smallest mipmap in the chain.

Texture Wrapping

[This is preliminary documentation and subject to change.]

This section presents information on wrapping textures around 3-D primitives. The discussion is divided into the following topics:

- What is Texture Wrapping?
- Using Texture Wrapping

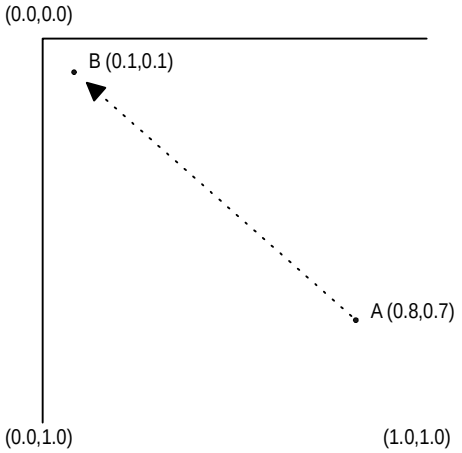
Note

Texture wrapping should not be confused with the similarly named texture addressing modes. For more information, see Texture Addressing Modes and Texture Wrapping and Texture Addressing Modes.

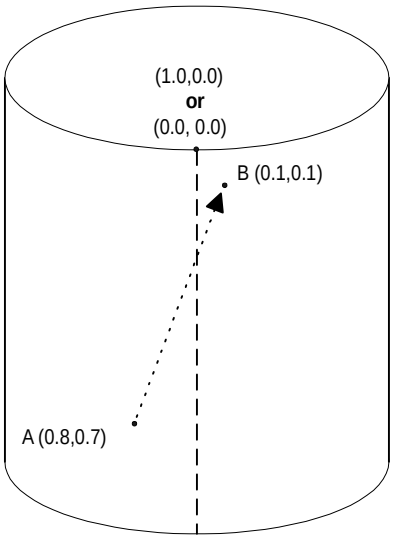
What is Texture Wrapping?

[This is preliminary documentation and subject to change.]

Texture wrapping, in short, changes the basic way in which Direct3D rasterizes textured polygons using the texture coordinates specified for each vertex. (Don't confuse texture wrapping with the "wrap" texture addressing mode. For more information, see Texture Addressing Modes and Texture Wrapping.) While rasterizing a polygon, the system interpolates between the texture coordinates at each of the polygon's vertices to determine the texels that should be used for every pixel of the polygon. Normally, the system treats the texture as a 2-D plane, interpolating new texels by taking the shortest route from point A within a texture to point B. If point A represents the U, V position (0.8, 0.1), and point B is at (0.1,0.1), the line of interpolation would look like:

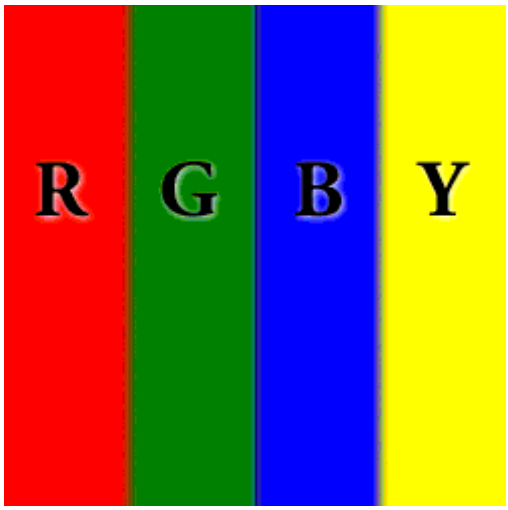


Note that the shortest distance between A and B in the preceding illustration runs roughly through the middle of the texture. Enabling U or V texture coordinate wrapping changes how Direct3D perceives the shortest route between texture coordinates in the U and V directions. By definition, texture wrapping causes the rasterizer to take the shortest route between texture coordinate sets, assuming that 0.0 and 1.0 are coincident. The last bit is the tricky part: you can imagine that enabling texture wrapping in one direction causes the system to treat a texture as though it were "wrapped" around a cylinder. For example, take the following illustration:

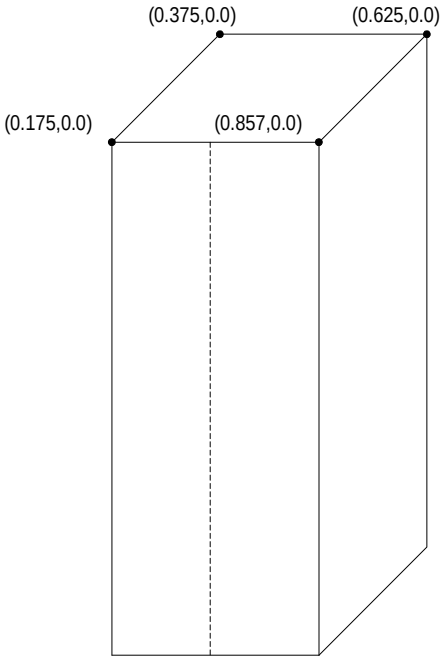


The preceding diagram shows how wrapping in the U direction affects the way the system interpolates texture coordinates. Using the same points we used in the example for "normal", or non-wrapped, textures, you can see that the shortest route between points A and B is no longer across the middle of the texture; it's now across the border where 0.0 and 1.0 exist together. Wrapping in the V direction is similar, only it "wraps" the texture around a cylinder that is lying on its side. Wrapping in both the U and V directions is a little more complex. In this situation, you might envision the texture as a torus, or doughnut.

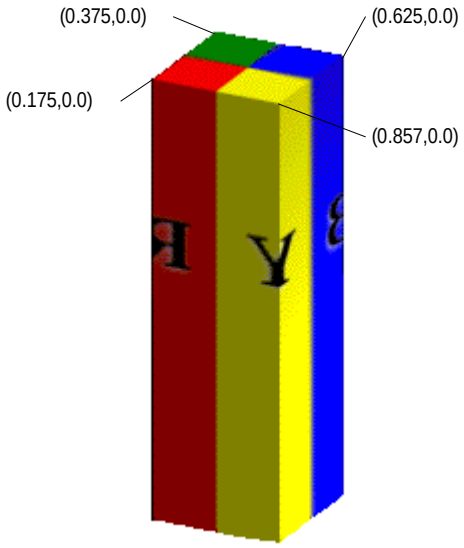
The most common practical application for texture wrapping is to perform environment mapping. Usually, an object textured with an environment map appears very reflective, showing a mirrored image of the object's surroundings in the scene. For the sake of this discussion, picture a room with four walls, each one painted with a letter R, G, B, Y and the corresponding colors: red, green, blue, and yellow. The environment map for such a simple room might look like:



Imagine that the room's ceiling is held up by a perfectly reflective, four sided, pillar. Mapping the environment map texture to the pillar is pretty simple — making it look as though it's reflecting the letters and colors as they appear on the walls isn't as easy. The following diagram shows a wire frame of the pillar with the applicable texture coordinates listed near the top vertices (the "seam" where wrapping will cross the edges of the texture is shown with a dotted line):



With wrapping enabled in the U direction, the textured pillar shows the colors and symbols from the environment map appropriately and, at the "seam" in the front of the texture, the rasterizer properly chooses the shortest route between the texture coordinates, assuming that U coordinates 0.0 and 1.0 share the same location. The textured pillar would look something like the following:



If texture wrapping wasn't enabled, the rasterizer would not interpolate in the direction needed to generate a believable, reflected, image. Rather, the area at the front of the pillar would contain a horizontally compressed version of the texels between U coordinates 0.175 and 0.875, as they pass through the center of the texture. The effect would be ruined.

Using Texture Wrapping

[This is preliminary documentation and subject to change.]

The process of enabling texture wrapping differs across versions the Direct3D device interfaces. If your application uses the **IDirect3DDevice3** interface, you enable texture wrapping for texture coordinate sets used by vertices, not for the texture stages themselves. In this case, call the **IDirect3DDevice3::SetRenderState** method to enable texture wrapping, passing one of the D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7 enumerated values as the first parameter to identify which texture coordinate set will receive wrapping. Specify the D3DWRAP_U and D3DWRAP_V flags in the second parameter to enable texture wrapping in the corresponding direction, or combine the two to enable wrapping in both directions. If you omit one or both of the flags, texture wrapping in the

corresponding direction is disabled. To disable texture wrapping for a particular set of texture coordinates, set the value for the corresponding render state to 0.

If your application uses the legacy **IDirect3DDevice2** or **IDirect3DDevice** interfaces, you still enable texture wrapping by calling the **SetRenderState** method; however, these interface versions do not support D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7. Instead, use the D3DRENDERSTATE_WRAPU or D3DRENDERSTATE_WRAPV value in the first parameter. Specify TRUE as in the second parameter to enable wrapping, or FALSE to disable it.

Note

The **IDirect3DDevice3** interface does recognize the legacy D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV render states, even though they were superseded by D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7. These older render states, when passed to the **IDirect3DDevice3** version of **SetRenderState**, affect U and V texture wrapping for the first set of texture coordinates.

Texture Blending

[This is preliminary documentation and subject to change.]

Direct3D can produce transparency effects by blending a texture with a primitive's color. It can also blend multiple textures onto a primitive. This section presents information on how texture blending is done. It is divided into the following topics:

- Alpha Texture Blending
- Multipass Texture Blending
- Multiple Texture Blending
- Light Mapping With Textures

If you want your application to use texture blending, the program should first check to see if the user's hardware supports it. The relevant information is found in the **dwTextureCaps** member of the **D3DPRIMCAPS** structure. For details on how to query the user's hardware for texture blending capabilities, see **IDirect3DDevice3::GetCaps** and **D3DDEVICEDESC**.

Alpha Texture Blending

[This is preliminary documentation and subject to change.]

When Direct3D renders a primitive, it generates a color for the primitive based on the primitive's material and lighting information. For details, see Lighting and Materials. If an application enables texture blending, Direct3D must then blend the texel colors of one or more textures with the primitive's current colors. Direct3D uses the following formula to determine the final color for each pixel in the primitive's image.

FinalColor = TexelColor * SourceBlendFactor +

PixelColor * DestBlendFactor

In the preceding formula, FinalColor is the pixel color that is output to the target rendering surface. TexelColor stands for the color of the texel that corresponds to the current pixel. For details on how Direct3D maps pixels to texels, see Texture Filtering. SourceBlendFactor is a calculated value that Direct3D uses to determine the percentage of the texel color to apply to the final color. PixelColor is the color of the current pixel in the primitive's image. DestBlendFactor represents the percentage of the current pixel's color that will be used in the final color. The values of SourceBlendFactor and DestBlendFactor range from 0.0 or 1.0 inclusive.

As you can see from the preceding formula, a texture is not rendered as transparent at all if the SourceBlendFactor is 1.0 and the DestBlendFactor is 0.0. It is completely transparent if the SourceBlendFactor is 0.0 and the DestBlendFactor is 1.0. If an application sets these factors to any other values, the resulting texture will be blended with some degree of transparency.

Every texel in a texture has a red, a green, and a blue color value. By default, Direct3D uses the alpha values of texels as the SourceBlendFactor. Therefore, applications can control the transparency of textures by setting the alpha values in their textures.

Your application can control the blending factors with the D3DRENDERSTATE_SRCBLEND and D3DRENDERSTATE_DESTBLEND enumerated values. Invoke the **IDirect3DDevice3::SetRenderState** method and pass either D3DRENDERSTATE_SRCBLEND or D3DRENDERSTATE_DESTBLEND as the value of the first parameter. The second parameter must be a member of the **D3DBLEND** enumerated type.

Multipass Texture Blending

[This is preliminary documentation and subject to change.]

Direct3D applications can achieve numerous special effects by applying many textures onto a primitive in more than one pass. The common term for this is multipass texture blending. A typical use for multipass texture blending is to apply shadows to primitives in addition to whatever textures they might normally have. For more information, see Light Mapping With Textures.

All Direct3D device interfaces support multipass texture blending. Beginning with the **IDirect3DDevice3** interface, DirectX is also able to apply multiple textures to primitives in a single pass, if the user's hardware supports it. For details, see Multiple Texture Blending.

If your user's hardware does not support multiple texture blending, your application can use multipass texture blending to achieve the same visual effects. However, it will not be able to sustain the frame rates that are possible when using multiple texture blending.

Applications enable multipass texture blending by invoking the **IDirect3DDevice3::SetRenderState** method, and passing the enumerated value

D3DRENDERSTATE_ALPHABLENDENABLE as the value of the first parameter. Pass TRUE as the value of the second parameter to enable texture blending, or FALSE to disable it.

Once texture blending is enabled, your application should set the source and destination blending factors based on the effect you want to achieve. For more information on the source and destination blending factors, see Alpha Texture Blending. Applications control the source blending factors by calling the **IDirect3DDevice3::SetRenderState** and passing it the enumerated value D3DRENDERSTATE_SRCBLEND as the value of the first parameter. Your program can set the destination blending factor by passing D3DRENDERSTATE_DESTBLEND as the value of the first parameter to **IDirect3DDevice3::SetRenderState**. In either case, the second parameter must be a member of the **D3DBLEND** enumerated type.

Applications set the blending operation by passing D3DRENDERSTATE_TEXTUREMAPBLEND as the value of the first parameter in a call to **IDirect3DDevice3::SetRenderState**. The second parameter must be a member of the **D3DTEXTUREBLEND** enumerated type.

On each pass, your application must set the current texture. Its texel colors will be blended with the existing pixel colors in the frame buffer. For information on setting the current texture, see Current Texture.

Multiple Texture Blending

[This is preliminary documentation and subject to change.]

With the introduction of the **IDirect3D3** and **IDirect3DDevice3** interfaces, Direct3D can now blend as many as eight textures onto primitives in a single pass. The use of multiple texture blending can profoundly increase the frame rate of Direct3D applications. Applications employ multiple texture blending to apply textures, shadows, specular lighting, diffuse lighting, and other special effects in a single pass.

To blend multiple textures, applications assign textures into the set of current textures, and then create blending stages. The topics in the following list present information on how these steps are accomplished:

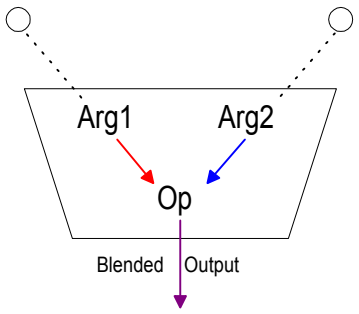
- Texture Stages and the Texture Blending Cascade
- Texture Blending Operations and Arguments
- Assigning the Current Textures
- Creating Blending Stages
- Legacy Blending Modes and Texture Stages

Texture Stages and the Texture Blending Cascade

[This is preliminary documentation and subject to change.]

Direct3D supports single-pass multiple texture blending through the use of "texture stages." A texture stage takes two arguments and performs a blending operation on

them, passing the result on for further processing or for rasterization. You could visualize a texture stage as shown in the following figure.

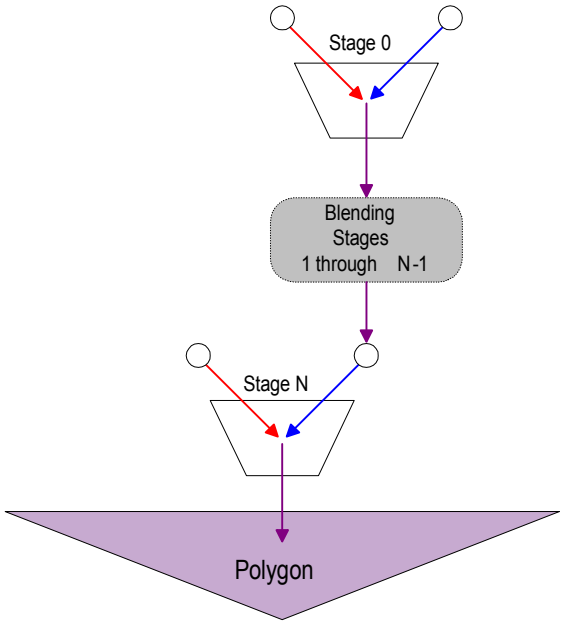


As the preceding illustration shows, texture stages blend two arguments by using a specified operator. Common operations include simple modulation or addition of the color or alpha components of the arguments, but more than two dozen operations are currently supported. The arguments for a stage can be an associated texture, the iterated color or alpha (iterated during Gouraud shading), arbitrary color and alpha, or the result from the previous texture stage. For more information, see Texture Blending Operations and Arguments.

Note

Direct3D distinguishes color blending from alpha blending. Applications set blending operations and arguments for color and alpha individually, and the results of those settings are independent of one another.

The combination of arguments and operations used by multiple blending stages define a simple flow-based blending language. The results from one stage flow down to another stage, and then from that stage to the next, and so on. The idea that results flow from stage to stage to eventually be rasterized on a polygon is often called the "texture blending cascade." The following illustration shows how individual texture stages make up the texture blending cascade.



Each stage in a device has a zero-based index. Direct3D for DirectX 6.0 allows up to eight blending stages, although you should always check device capabilities to determine how many stages the current hardware supports. The first blending stage is at index 0, the second is at 1, and so on up to index 7. The system blends stages in increasing order of index.

Use only the number of stages you need; the unused blending stages are disabled by default. So, if your application only uses the first two stages, it need only set operations and arguments for stage 0 and 1, leaving the remaining stages alone. The system blends the two stages, and ignores the disabled stages.

Optimization Note

If your application varies the number of stages it uses for different situations — such as four stages for some objects, and only two for others — you don't need to explicitly disable all previously used stages. If you disable the color operation for the first unused stage, all stages with a higher index will not be applied. You can disable texture mapping altogether by setting the color operation for the first texture stage (stage 0).

Texture Blending Operations and Arguments

[This is preliminary documentation and subject to change.]

Applications associate a blending stage with each texture in the set of current textures. As mentioned in Texture Stages and the Texture Blending Cascade, Direct3D evaluates each blending stage in order, beginning with the first texture in the set and ending with the eighth.

Direct3D applies the information from each texture in the set of current textures to the blending stage that is associated with it. Applications control what information from a given texture stage is used by calling **IDirect3DDevice3::SetTextureStageState**. You can set separate operations for the color and alpha channels independently, and each operation uses two arguments. Specify color channel operations by using the D3DTSS_COLOROP stage state, and D3DTSS_ALPHAOP for alpha operations– both use values from the **D3DTEXTUREOP** enumerated type.

Texture blending arguments use the D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHARG1, and D3DTSS_ALPHARG2 members of the **D3DTEXTURESTAGESTATETYPE** enumerated type. The corresponding argument values are identified using texture argument flags.

Note

You can disable a texture stage — and any subsequent texture blending stages in the cascade — by setting the color operation for that stage to D3DTOP_DISABLE. Disabling the color operation effectively disables the alpha operation as well.

Assigning the Current Textures

[This is preliminary documentation and subject to change.]

Direct3D maintains a list of up to eight current textures. It blends these textures onto all of the primitive it renders. Only textures created as texture interface pointers can be used in the set of current textures.

Applications call the **IDirect3DDevice3::SetTexture** method to assign textures into the set of current textures. The first parameter must be from the a number in the range of 0-7 inclusive. Pass the texture interface pointer as the second parameter.

The following code fragment demonstrates how a texture can be assigned into the set of current textures:

```
// This code fragment assumes that the variable lpd3dDev is a valid
// pointer to an IDirect3D3 interface and lpd3dTexture is a valid
// pointer to an IDirect3DTexture2 interface.

// Set the third texture.
lpd3dDev->SetTexture(2, lpd3dTexture);
```

Note

Software devices do not support assigning a texture to more than one texture stage at a time.

Creating Blending Stages

[This is preliminary documentation and subject to change.]

A blending stage is a set of texture operations, together with their arguments, that define how textures are blended. When making a blending stage, applications invoke the **IDirect3DDevice3::SetTextureStageState** function. The first call specifies the operation that will be performed. Two additional invocations define the arguments to which the operation will be applied. The following code fragment illustrates the creation of a blending stage:

```
// This example assumes that lpD3DDev is a valid pointer to an
// IDirect3DDevice3 interface.

// Set the operation for the 1st texture.
lpD3DDev->SetTextureStageState(0,D3DTSS_COLOROP,D3DTOP_ADD);

// Set arg1 for the texture operation.
lpD3DDev->SetTextureStageState(0,               // First texture
                       D3DTSS_COLORARG1,  // Set color arg 1
                       D3DTA_TEXTURE);    // Color arg 1 value

// Set arg2 for the texture operation.
lpD3DDev->SetTextureStageState(0,               // First texture
                       D3DTSS_COLORARG2,  // Set color arg 2
                       D3DTA_DIFFUSE);    // Color arg 2 value
```

Texel data in textures contain color and alpha values. Programs can define separate operations for both color and alpha values in a single blending stage. Each operation (color and alpha) has its own arguments. For details, see **D3DTEXTURESTAGESTATETYPE**.

Although not part of the Direct3D API, the following macros can also be inserted into your program to abbreviate the code required for creating texture blending stage.

```
#define SetTextureColorStage( dev, i, arg1, op, arg2 )      \
  dev->SetTextureStageState( i, D3DTSS_COLOROP, op);       \
  dev->SetTextureStageState( i, D3DTSS_COLORARG1, arg1 );   \
  dev->SetTextureStageState( i, D3DTSS_COLORARG2, arg2 );

#define SetTextureAlphaStage( dev, i, arg1, op, arg2 )      \
  dev->SetTextureStageState( i, D3DTSS_ALPHAOP, op);       \
  dev->SetTextureStageState( i, D3DTSS_ALPHARG1, arg1 );    \
  dev->SetTextureStageState( i  D3DTSS_ALPHARG2, arg2 );
```

Legacy Blending Modes and Texture Stages

[This is preliminary documentation and subject to change.]

Although Direct3D still supports the texture blending render state, D3DRENDERSTATE_TEXTUREMAPBLEND, the blending modes it offers should not be used in combination with texture stage based texture blending, as the results

can be unpredictable. However, you can "build" your own equivalents to the legacy blending modes by using texture stages.

This list shows the legacy blending modes (identified by the members of the **D3DTEXTUREBLEND** enumerated type), followed by a short example that sets up the same blend by way of texture stage states. (For all examples, the *g_lpDev* variable is assumed to be a valid pointer to an **IDirect3DDevice3** interface):

D3DTBLEND_ADD

```
g_lpDev->SetTextureStageState(0, COLOROP, D3DTOP_ADD);
g_lpDev->SetTextureStageState(0, COLORARG1, D3DTA_TEXTURE);
g_lpDev->SetTextureStageState(0, COLORARG2, D3DTA_DIFFUSE);

g_lpDev->SetTextureStageState(0, ALPHAOP, D3DTOP_SELECTARG2);
g_lpDev->SetTextureStageState(0, ALPHAARG2, D3DTA_DIFFUSE);
```

D3DTBLEND_COPY and D3DTBLEND_DECAL

```
g_lpDev->SetTextureStageState(0, COLOROP, D3DTOP_SELECTARG1);
g_lpDev->SetTextureStageState(0, COLORARG1, D3DTA_TEXTURE);

g_lpDev->SetTextureStageState(0, ALPHAOP, D3DTOP_SELECTARG1);
g_lpDev->SetTextureStageState(0, ALPHAARG2, D3DTA_TEXTURE);
```

D3DTBLEND_DECALALPHA

```
g_lpDev->SetTextureStageState(0, COLOROP, D3DTOP_BLENDTEXTUREALPHA);
g_lpDev->SetTextureStageState(0, COLORARG1, D3DTA_TEXTURE);
g_lpDev->SetTextureStageState(0, COLORARG2, D3DTA_DIFFUSE);

g_lpDev->SetTextureStageState(0, ALPHAOP, D3DTOP_SELECTARG2);
g_lpDev->SetTextureStageState(0, ALPHAARG2, D3DTA_DIFFUSE);
```

D3DTBLEND_MODULATE

```
g_lpDev->SetTextureStageState(0, COLOROP, D3DTOP_MODULATE);
g_lpDev->SetTextureStageState(0, COLORARG1, D3DTA_TEXTURE);
g_lpDev->SetTextureStageState(0, COLORARG2, D3DTA_DIFFUSE);

if ( the_texture_has_an_alpha_channel )
{
   g_lpDev->SetTextureStageState(0, ALPHAOP, D3DTOP_SELECTARG1);
   g_lpDev->SetTextureStageState(0, ALPHAARG1, D3DTA_TEXTURE);
}
else
{
   g_lpDev->SetTextureStageState(0, ALPHAOP, D3DTOP_SELECTARG2);
   g_lpDev->SetTextureStageState(0, ALPHAARG2, D3DTA_DIFFUSE);
}
```

D3DTBLEND_MODULATEALPHA

```
g_lpDev->SetTextureStageState(0, COLOROP, D3DTOP_MODULATE);
g_lpDev->SetTextureStageState(0, COLORARG1, D3DTA_TEXTURE);
g_lpDev->SetTextureStageState(0, COLORARG2, D3DTA_DIFFUSE);

g_lpDev->SetTextureStageState(0, ALPHAOP, D3DTOP_MODULATE);
g_lpDev->SetTextureStageState(0, ALPHAARG1, D3DTA_TEXTURE);
g_lpDev->SetTextureStageState(0, ALPHAARG2, D3DTA_DIFFUSE);
```

D3DTBLEND_DECALMASK and D3DTBLEND_MODULATEMASK
Not supported in DirectX 6.0.

Light Mapping With Textures

[This is preliminary documentation and subject to change.]

For an application to realistically render a 3-D scene, it must take into account the effect that light sources have on the appearance of the scene. Although techniques such as flat and Gouraud shading are valuable tools in this respect, they can be insufficient for your needs. Direct3D supports multipass and multiple texture blending. These capabilities enable your application to render scenes whose appearance is much more realistic than scenes rendered with shading techniques alone. By applying one or more light maps, your application can map areas of light and shadow onto its primitives.

A light map is a texture or group of textures that contain information about lighting in a 3-D scene. You can store the lighting information in the alpha values of the light map, in the color values, or in both.

If you implement light mapping using multipass texture blending, your program should render the light map onto its primitives on the first pass. It should use a second pass to render the base texture. The exception to this is specular light mapping. In that case, render the base texture first, then add the light map.

Multiple texture blending enables your application to render both the light map and the base texture in one pass. If your user's hardware provides for multiple texture blending, your application should take advantage of it when performing light mapping. It will significantly improve your application's performance.

Using light maps, a Direct3D application can achieve a variety of lighting effects when it renders primitives. It can not only map monochrome and colored lights in a scene, it can add details such as specular highlights and diffuse lighting.

Information on using Direct3D texture blending to perform light mapping is presented in the following topics:

- Monochrome Light Maps
- Color Light Maps
- Specular Light Maps
- Diffuse Light Maps

Monochrome Light Maps

[This is preliminary documentation and subject to change.]

Some older 3-D accelerator boards do not support texture blending using the alpha value of the destination pixel (see Alpha Texture Blending). These adapters also generally do not support multiple texture blending. If your application is running on an adapter such as this, it can use multipass texture blending to perform monochrome light mapping.

To perform monochrome light mapping, an application stores the lighting information in the alpha data of its light map textures. The program uses the texture filtering capabilities of Direct3D to perform a mapping from each pixel in the primitive's image to a corresponding texel in the light map. It sets the source blending factor to the alpha value of the corresponding texel.

The following code fragment illustrates how an application can use a texture as a monochrome light map:

```
// This example assumes that lpD3DDev is a valid pointer to an
// IDirect3DDevice3 interface and that lptexLightMap is a valid
// pointer to a texture that contains monochrome light map data.

// Set the light map texture as the current texture.
lpD3DDev->SetTexture(0,lptexLightMap);

// Set the color operation.
lpD3DDev->SetTextureStageState(0,D3DTSS_COLOROP,
                    D3DTOP_SELECTARG1);

// Set argument 1 to the color operation.
lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG1,
                    D3DTA_TEXTURE | D3DTA_ALPHAREPLICATE);
```

Since display adapters that do not support destination alpha blending usually do not support multiple texture blending, this example sets the light map as the first texture, which is available on all 3-D accelerator cards. The sample code sets the color operation for the texture's blending stage to blend the texture data with the primitive's existing color. It then selects the first texture and the primitive's existing color as the input data.

Color Light Maps

[This is preliminary documentation and subject to change.]

Your application will usually render 3-D scenes more realistically if it uses colored light maps. A colored light map uses the RGB data in the light map for its lighting information. The following code fragment demonstrates light mapping with RGB color data:

```
// This example assumes that lpD3DDev is a valid pointer to an
```

```
// IDirect3DDevice3 interface and that lptexLightMap is a valid
// pointer to a texture that contains RGB light map data.

// Set the light map texture as the 1st texture.
lpD3DDev->SetTexture(0, lptexLightMap);

lpD3DDev->SetTextureStageState(0,D3DTSS_COLOROP,
                    D3DTOP_MODULATE);

lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG1,
                    D3DTA_TEXTURE);

lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG2,
                    D3DTA_DIFFUSE);
```

This sample sets the light map as the first texture. It then sets state of the first blending stage to modulate the incoming texture data. It uses the first texture and the current color of the primitive as the arguments to the modulate operation.

Specular Light Maps

[This is preliminary documentation and subject to change.]

When illuminated by a point light source, shiny surfaces, such as metal or color light maps, your Direct3D applications can apply specular light maps to primitives.

To perform specular light mapping, first modulate the specular light map with the primitive's existing texture. Then add the monochrome or RGB light map. The following code fragment illustrates this process:

```
// This example assumes that lpD3DDev is a valid pointer to an
// IDirect3DDevice3 interface.
// lptexBaseTexture is a valid pointer to a texture.
// lptexSpecLightMap is a valid pointer to a texture that contains RGB
// specular light map data.
// lptexLightMap is a valid pointer to a texture that contains RGB
// light map data.

// Set the base texture.
lpD3DDev->SetTexture(0,lptexBaseTexture );

// Set the base texture operation and args
lpD3DDev->SetTextureStageState(0,D3DTSS_COLOROP,
                    D3DTOP_MODULATE );
lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// Set the specular light map.
```

```
lpD3DDev->SetTexture(1,lptexSpecLightMap );

// Set the specular light map operation and args
lpD3DDev->SetTextureStageState(1,D3DTSS_COLOROP,
                    D3DTOP_MODULATE );
lpD3DDev->SetTextureStageState(1,D3DTSS_COLORARG1, D3DTA_TEXTURE );
lpD3DDev->SetTextureStageState(1,D3DTSS_COLORARG2, D3DTA_CURRENT );

// Set the RGB light map.
lpD3DDev->SetTexture(2,lptexLightMap);

// Set the RGB light map operation and args
lpD3DDev->SetTextureStageState(2,D3DTSS_COLOROP, D3DTOP_ADD);
lpD3DDev->SetTextureStageState(2,D3DTSS_COLORARG1, D3DTA_TEXTURE );
lpD3DDev->SetTextureStageState(2,D3DTSS_COLORARG2, D3DTA_CURRENT );
```

Diffuse Light Maps

[This is preliminary documentation and subject to change.]

When illuminated by a point light source, matte surfaces display diffuse light reflection. The brightness of diffuse light depends on the distance from the light source and the angle between the surface normal and the light source direction vector.

Your application can simulate diffuse lighting with texture light maps. Do this by adding the diffuse light map to the base texture, as shown in the following code fragment:

```
// This example assumes that lpD3DDev is a valid pointer to an
// IDirect3DDevice3 interface.
// lptexBaseTexture is a valid pointer to a texture.
// lptexDiffuseLightMap is a valid pointer to a texture that contains
// RGB diffuse light map data.

// Set the base texture.
lpD3DDev->SetTexture(0,lptexBaseTexture );

// Set the base texture operation and args
lpD3DDev->SetTextureStageState(0,D3DTSS_COLOROP,
                    D3DTOP_MODULATE );
lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG1, D3DTA_TEXTURE );
lpD3DDev->SetTextureStageState(0,D3DTSS_COLORARG2, D3DTA_DIFFUSE );

// Set the diffuse light map.
lpD3DDev->SetTexture(1,lptexDiffuseLightMap );

// Set the blend stage.
lpD3DDev->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_ADD );
```

```
lpD3DDev->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_TEXTURE );
lpD3DDev->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_CURRENT );
```

Texture Compression

[This is preliminary documentation and subject to change.]

DirectDraw provides services to compress surfaces that will be used for texturing 3-D models. For information on creating and manipulating the data in a compressed texture surface, see Compressed Texture Surfaces in the DirectDraw documentation.

Once your application creates a rendering device, it can determine if the device supports texturing from compressed texture surfaces by calling the **IDirect3DDevice3::EnumTextureFormats** method. The method calls the **D3DEnumPixelFormatsCallback** callback function that you provide for each pixel format that the device supports for texture maps. If any of the enumerated pixel formats use the DXT1, DXT2, DXT3, DXT4, or DXT5 four character codes (FOURCCs), the device can texture directly from a compressed texture surface that uses that format. If so, you can use compressed texture surfaces directly with Direct3D by calling the **IDirect3DDevice3::SetTexture** method. If the device doesn't support texturing from compressed texture surfaces, you can still store texture data in a compressed format surface, but you must convert any compressed textures to a supported format before they can be used for texturing. The DirectDraw blit methods, **IDirectDrawSurface4::Blt** and **IDirectDrawSurface4::BltFast**, automatically perform conversion of compressed formats to standard RGBA formats. To minimize loss of information, try to pick a destination pixel format that closely matches the compressed format. For instance, ARGB:1555 would be a good destination format for DXT1, but ARGB:4444 would be a better choice for DXT3 since DXT3 contains four bits of alpha information.

Automatic Texture Management

[This is preliminary documentation and subject to change.]

Texture management, in short, is the process of determining which textures are needed for rendering at a given time, and ensuring that those textures are loaded into video memory. Like any algorithm, texture management schemes vary in complexity, but any approach to texture management involves the following key tasks:

- Tracking the amount of available texture memory.
- Calculating which textures are currently needed for rendering, and which aren't.
- Determining which of the existing texture surfaces can be reloaded with another texture image, and which surfaces should be destroyed and replaced with new texture surfaces.

Previously, Direct3D applications were responsible for managing textures on their own. Direct3D Immediate Mode for DirectX 6.0 introduces system supported texture

management, where Direct3D efficiently and intelligently performs texture management, ensuring that textures are loaded for optimal performance. (Texture surfaces that Direct3D manages are casually referred to as "managed textures.")

You request automatic texture management for textures when you create them. To get a managed texture, simply create a texture surface that also includes the DDSCAPS2_TEXTUREMANAGE flag in the **dwCaps2** member of the associated **DDSCAPS2** structure. Note that you are not allowed to specify where you want the texture created–you can't use the DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY flags when creating a managed texture. After creating the managed texture, you can call the **IDirect3DDevice3::SetTexture** method to set it to a stage in the rendering device's texture cascade.

Direct3D automatically downloads textures into video memory as needed. (The system might cache managed textures in local or non-local video memory, depending on the availability of non-local video memory or other factors. Where your managed textures are cached is not communicated to your application, nor is this knowledge required to take advantage of automatic texture management.) If your application uses more textures than can fit in video memory, Direct3D will evict older textures from video memory to make room for the new textures. If you use an evicted texture again, the system uses the original system memory texture surface to reload the texture in the video memory cache. Reloading the texture is a minor, but obviously necessary, performance hit to the application.

You can dynamically modify the original system memory copy of the texture by blitting or locking the texture surface. When the system detects a "dirty" surface — after a blit is completed, or when the surface is unlocked — the texture manager automatically updates the video memory copy of the texture. The performance hit incurred is similar to reloading an evicted texture.

When entering a new level in a game, your application may need to flush all managed textures from video memory. You can explicitly request that all managed textures be evicted by calling the **IDirect3D3::EvictManagedTextures** method. When you call this method, Direct3D destroys any cached local and non-local video memory textures, but leaves the original system memory copies untouched.

Note

You cannot retrieve a texture handle for a managed texture by calling the **IDirect3DTexture2::GetHandle** method.

Hardware Considerations for Texturing

[This is preliminary documentation and subject to change.]

Current hardware does not necessarily implement all of the functionality that the Direct3D interface enables. Your program must test user's hardware and adjust its rendering strategies accordingly.

Many 3-D accelerator cards do not support diffuse iterated values as arguments to blending units. However, your program can introduce iterated color data when it performs texture blending.

Some 3-D hardware may not have a blending stage associated with the first texture. On these adapters, your application will need to perform blending in the second and third texture stages in the set of current textures.

Due to limitations in much of today's hardware, few display adapters can perform trilinear mipmap interpolation through the multiple texture blending interface offered by **IDirect3DDevice3**. Specifically, there is little support for setting the D3DTSS_MIPFILTER texture stage state to D3DTFP_LINEAR. Your application can use multipass texture blending to achieve the same effects, or degrade to the D3DTFP_POINT mipmap filter mode, which is widely supported.

Depth Buffers

[This is preliminary documentation and subject to change.]

This section presents information on using depth buffers for hidden line and hidden surface removal. It is organized into the following topics:

- What Are Depth Buffers?
- Using Depth Buffers

The Direct3D Immediate Mode tutorials provide additional information about using depth buffers. See Tutorial 2: Adding a Depth Buffer for details.

What Are Depth Buffers?

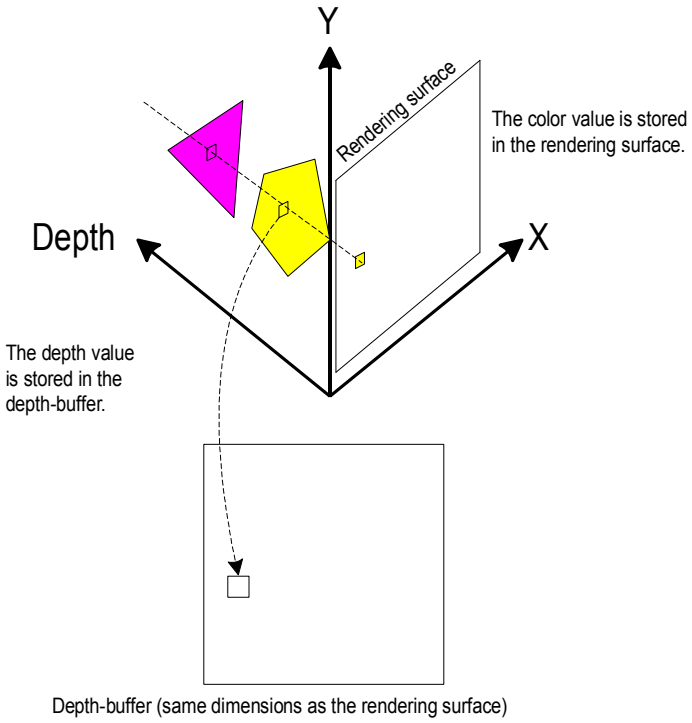
[This is preliminary documentation and subject to change.]

A depth buffer, often called a z-buffer or a w-buffer, is a DirectDraw surface that stores depth information to be used by Direct3D. When Direct3D renders a 3-D scene to a target surface, it can use the memory in an attached depth buffer surface as a workspace to determine how the pixels of rasterized polygons occlude one another. Direct3D uses an off-screen DirectDraw surface as the target to which final color values are written. The depth buffer surface that is attached to the render target surface is used to store depth information which tells Direct3D how "deep" each visible pixel is in the scene.

When a 3-D scene is rasterized with depth buffering enabled, each point on the rendering surface is tested. The values in the depth buffer can be a point's z coordinate or its homogeneous w coordinate — from the point's (x,y,z,w) location in projection space. A depth buffer that uses z values is often called a "z-buffer," and one that uses w values called a "w-buffer." Each type of depth buffer has its advantages and disadvantages, which are discussed later.

At the beginning of the test, the depth value in the depth buffer is set to the largest possible value for the scene. The color value on the rendering surface is set to either

the background color value, or the color value of the background texture at that point. Each polygon in the scene is tested to see if it intersects with the current coordinate (x,y) on the rendering surface. If it does, the depth value (which will be the z coordinate in a z-buffer, and the w coordinate in a w-buffer) at the current point is tested to see if it is smaller than the depth value already stored in the depth buffer. If the depth of the polygon value is smaller, it is stored in the depth buffer and the color value from the polygon is written to the current point on the rendering surface. If the depth value of the polygon at that point is larger, the next polygon in the list is tested. This process is shown in the following illustration.



Note

Although most applications don't use this feature, you can change the comparison Direct3D uses to determine which values will be placed in the depth buffer and subsequently the render target surface. To do so, change the value for the D3DRENDERSTATE_ZFUNC render state.

Nearly all 3-D accelerators on the market support z-buffering, making z-buffers the most common type of depth buffer today. However ubiquitous, z-buffers do have their drawbacks. Due to the mathematics involved, the generated z values in a z-buffer tend not to be distributed evenly across the z-buffer range (typically 0.0 to 1.0, inclusive). Specifically, the ratio between the far and near clipping planes strongly affects how unevenly z values are distributed. Using a far-plane distance to near-plane distance ratio of 100, 90% of the depth buffer range is spent on the first 10% of the scene depth range. Typical applications for entertainment or visual simulations with

exterior scenes often require far plane/near plane ratios of anywhere between 1000 to 10000. At a ratio of 1000, 98% of the range is spent on the 1st 2% of the depth range, and the distribution gets worse with higher ratios. This can cause hidden surface artifacts in distant objects, especially when using 16-bit depth buffers (the most commonly supported bit-depth).

A w-based depth buffer, on the other hand, is often more evenly distributed between the near and far clip planes than z-buffer. The key benefit is that the ratio of distances for the far and near clipping planes is no longer an issue. This allows applications to support large maximum ranges, while still getting relatively accurate depth buffering close to the eye point. A w-based depth buffer isn't perfect, and can sometimes exhibit hidden surface artifacts for near objects. Another drawback to the w-buffered approach is related to hardware support: w-buffering isn't supported as widely in hardware as z-buffering.

The DirectDraw HEL can create depth buffers for use by Direct3D or other 3-D–rendering software. The HEL supports 16-bit depth buffers. The DirectDraw device driver for a 3-D accelerator can permit the creation of depth buffers in display memory by exposing the DDSCAPS_ZBUFFER flag. You can query for the supported depth buffer bit-depths by calling the **IDirect3D3::EnumZBufferFormats** method.

Whenever a z-buffer surface is created, your application should maintain a pointer to the z-buffer until it shuts the Direct3D system down. Your program should release the z-buffer surface just before it releases the rendering surface.

Z-buffering requires overhead during rendering. Various techniques can be used to optimized rendering when using z-buffers. For details, see Z-Buffer Performance.

Note

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

Using Depth Buffers

[This is preliminary documentation and subject to change.]

The following topics discuss common usage scenarios for depth buffers:

- Querying for Depth Buffer Support
- Creating a Depth Buffer
- Enabling Depth Buffering
- Clearing Depth Buffers
- Changing Depth Buffer Write Access
- Changing Depth Buffer Comparison Functions
- Using Z-Bias

For a conceptual overview and an introduction to depth buffering, see What Are Depth Buffers?. For more information about using depth-buffers in your applications, see Tutorial 2: Adding a Depth Buffer.

Querying for Depth Buffer Support

[This is preliminary documentation and subject to change.]

As with any feature, don't assume that the driver your application uses supports depth buffering; you should always check the driver's capabilities. Although most driver support z-based depth buffering, not all will be able to provide support to w-based depth buffering. (For general information about depth buffering, see What Are Depth Buffers?) Drivers do not fail if you attempt to enable an unsupported scheme, falling-back on another depth buffering method instead, or sometimes disabling depth buffering altogether, which can result in rendered scenes that show major depth sorting artifacts.

You can check for general support for depth buffers by querying the DirectDraw for the display device your application uses before you create a Direct3D device. If the DirectDraw object reports that it supports depth buffering, any hardware devices you create from this DirectDraw object will support z-buffering (but you don't yet know if the driver supports w-buffering).

Û To query for general depth buffering support

1. Call the **IDirectDraw4::GetCaps** method of the DirectDraw object for the display device your application uses, passing initialized **DDCAPS** structures as parameters. After the call, the **DDCAPS** structures contain information about DirectDraw's hardware and emulation capabilities.
2. Examine the **ddsCaps** member of the structure you passed as the first parameter. If this member–a **DDSCAPS2** structure–includes the DDSCAPS_ZBUFFER flag, the driver supports depth buffering through z-buffers.

Once you know that the driver supports z-buffers, you can verify w-buffer support. Although z-buffers are supported for all software rasterizers, w-buffers are only supported by the reference rasterizer, which is hardly suited for use by real-world applications. No matter what type of device your application uses, you should verify support for w-buffers before you attempt to enable w-based depth buffering.

Û To determine support for w-buffers

1. After creating your device (HAL or emulated), call the **IDirect3DDevice3::GetCaps** method, passing initialized **D3DDEVICEDESC** structures in both parameters.
2. After the call, the **dpcTriCaps** and **dpcLineCaps** members (**D3DPRIMCAPS** structures) contain information about the driver's support for rendering primitives.
3. If the **dwRasterCaps** member of these structures contains the D3DPRASTERCAPS_WBUFFER flag, then the driver supports w-based depth buffering for that primitive type.

Creating a Depth Buffer

[This is preliminary documentation and subject to change.]

You usually create a depth buffer during your application's startup sequence, before creating a Direct3D device object for rendering. Use the following steps to create and attach a depth buffer surface to the render target surface:

Û To create a depth buffer

1. Call the **IDirect3D3::EnumZBufferFormats** method to determine the depth-buffer pixel formats that the device supports.
2. Prepare a **DDSURFACEDESC2** structure that describes a DirectDraw surface that matches the render target surface's dimensions, includes the DDSCAPS_ZBUFFER capability flag, and uses a supported depth-buffer pixel format (retrieved in Step 1).
3. Create the surface in video memory or system memory, depending on what type of rendering device your application will use. (See note.)
4. Attach the depth buffer surface to the rendering surface using the **IDirectDrawSurface4::AddAttachedSurface** method.

After creating the depth buffer surface and attaching it to the render target surface, call the **IDirect3D3::CreateDevice** method to create a rendering device that uses the render target surface and its depth buffer.

Create the depth buffer in video memory — with the DDSCAPS_VIDEOMEMORY surface capability — when your application uses a hardware driver (HAL device), and in system memory — the DDSCAPS_SYSTEMMEMORY surface capability — when using software emulation drivers (the MMX or RGB devices). Failing to create a depth buffer in the appropriate type of memory will cause the **CreateDevice** method to fail.

Note

Some popular hardware devices require that the render target and depth buffer surfaces use the same bit depth. On such hardware, if your application uses a 16-bit render target surface, the attached depth buffer must also be 16-bits. For a 32-bit render target surface, the depth buffer must be 32-bits, of which 8-bits can be used for stencil buffering (if needed).
If the hardware upon which your application is running has this requirement, and your application fails to meet it, any attempts to create a rendering device that uses the non-compliant surfaces will fail. You can use the DirectDraw method, **IDirectDraw4::GetDeviceIdentifier** to track hardware that imposes this limitation.

Enabling Depth Buffering

[This is preliminary documentation and subject to change.]

After you create a depth buffer (as described in Creating a Depth Buffer), enabling depth buffering is as simple as calling the **IDirect3DDevice3::SetRenderState** method. Set the D3DRENDERSTATE_ZENABLE render state to enable depth-buffering. Use the D3DZB_TRUE value (or TRUE) to enable z-buffering, D3DZB_USEW to enable w-buffering, or D3DZB_FALSE (or FALSE) to disable depth buffering.

Note

To use w-buffering, your application must set a compliant projection matrix even if it doesn't use the Direct3D transformation pipeline, and perspective-correct texture mapping must be enabled. For information about providing an appropriate projection matrix, see A W-Friendly Projection Matrix. (The projection matrix discussed in What Is the Projection Transformation? is compliant.) To enable perspective-correct texture mapping, set the D3DRENDERSTATE_TEXTUREPERSPECTIVE render state to TRUE. For DirectX 6.0 and later, this is the default value.

Clearing Depth Buffers

[This is preliminary documentation and subject to change.]

You should clear the depth buffer each time you render a new frame. You can explicitly clear the depth buffer through Direct3D by using the **IDirect3DViewport3::Clear** and **IDirect3DViewport3::Clear2** methods. The **Clear** method always uses the "deepest" value, but **Clear2** allows you to specify an arbitrary depth value.

You can also use DirectDraw to clear a depth buffer. Call the depth buffer surface's **IDirectDrawSurface4::Blt** method to clear it. The DDBLT_DEPTHFILL flag indicates that the blit is being used to a clear depth buffer. When this flag is specified, the **DDBLTFX** structure passed to the **IDirectDrawSurface4::Blt** method should be initialized and have its **dwFillDepth** member set to the required depth.

If the DirectDraw device driver for a 3-D accelerator is designed to provide support for depth buffer clearing in hardware, it will report the DDCAPS_BLTDEPTHFILL flag and should handle DDBLT_DEPTHFILL blits. The destination surface of a depth-fill blit must be a depth buffer surface.

Changing Depth Buffer Write Access

[This is preliminary documentation and subject to change.]

By default, the Direct3D system is allowed to write to the depth buffer. Most applications will leave writing to the depth buffer enabled, but there are some special effects that can be achieved by not allowing the Direct3D system to write to the depth buffer.

You can disable depth buffer writes by calling the **IDirect3DDevice3::SetRenderState** method with the *dwRenderStateType* parameter

set to D3DRENDERSTATE_ZWRITEENABLE and the *dwRenderState* parameter should be set to 0.

Changing Depth Buffer Comparison Functions

[This is preliminary documentation and subject to change.]

By default, when depth testing performed on a rendering surface, the Direct3D system updates the render target surface if the corresponding depth value (z or w) for each point is less than the value already in the depth buffer. You can change how the system performs comparisons on depth values by calling the **IDirect3DDevice3::SetRenderState** method with the *dwRenderStateType* parameter set to D3DRENDERSTATE_ZFUNC. The *dwRenderState* parameter should be set to one of the values in the **D3DCMPFUNC** enumeration.

Using Z-Bias

[This is preliminary documentation and subject to change.]

Polygons that are coplanar in your 3-D space can be made to appear as if they are not coplanar by adding a z-bias to each one. This is a technique commonly used to ensure that shadows in a scene are displayed properly. For instance, a shadow on a wall will likely have the same depth value as the wall does. If you render the wall first, then the shadow, the shadow might not be visible, or depth artifacts may be visible. You could reverse the order in which you render the coplanar objects in hopes of reversing the effect, but it depth artifacts are still likely.

You can help ensure that coplanar polygons are rendered properly by adding a bias to the z-values that the system uses when rendering the sets of coplanar polygons. To add a z-bias to a set of polygons, call the **IDirect3DDevice3::SetRenderState** method just before rendering them, setting the *dwRenderStateType* parameter to D3DRENDERSTATE_ZBIAS, and the *dwRenderState* parameter to a value between 0-16 inclusive. A higher z-bias value will increase the likelihood that the polygons you render will be visible when displayed with other, coplanar, polygons.

Stencil Buffers

[This is preliminary documentation and subject to change.]

This section presents a discussion of the purpose and use stencil buffers. It is divided into the following topics"

- What is a Stencil Buffer?
- How the Stencil Buffer Works
- Customizing the Stencil Buffer

What is a Stencil Buffer?

[This is preliminary documentation and subject to change.]

The stencil buffer enables or disables drawing to the rendering target surface on a pixel-by-pixel basis. At its most fundamental level, it enables applications to mask off sections of the rendered image so that it is not displayed. Applications often use stencil buffers for special effects such as dissolves, decaling, and outlining. For details, see Stencil Buffer Techniques.

Stencil buffer information is embedded in the z-buffer data. Your application can test the user's hardware to see if it supports stencil buffers by invoking the **IDirect3D3::EnumZBufferFormats** method. To obtain information about the particular format of the z-buffer data, call the **IDirectDrawSurface4::GetPixelFormat** method, which passes a **DDPIXELFORMAT** structure to your application. The relevant information is in the **dwZBufferBitDepth**, **dwStencilBitDepth**, **dwZBitMask**, and **dwStencilBitMask** members.

How the Stencil Buffer Works

[This is preliminary documentation and subject to change.]

Direct3D performs a test on the contents of the stencil buffer on a pixel-by-pixel basis. For each pixel in the target surface, it performs a test using the corresponding value in the stencil buffer, a stencil reference value, and a stencil mask value. If the test passes, Direct3D performs an action. The test is performed using the following steps:

1. Bitwise AND the stencil reference value with the stencil mask.
2. Bitwise AND the stencil buffer value for the current pixel with the stencil mask.
3. Compare the result of step 1 to the result of step 2 using the comparison function.

Written in pseudocode, these steps would look like this:

(StencilRef & StencilMask) CompFunc (StencilBufferValue & StencilMask)

Where StencilBufferValue is the contents of the stencil buffer for the current pixel. This pseudocode uses the & symbol to represent the bitwise **AND** operation. StencilMask represents the value of the stencil mask, and StencilRef represents the stencil reference value. CompFunc is the comparison function.

The current pixel is written to the target surface if the stencil test passes, and ignored otherwise. The default comparison behavior is to write the pixel no matter how each of the bitwise operations turn-out (D3DCMP_ALWAYS). You can change this behavior by changing the value of the D3DRENDERSTATE_STENCILFUNC render state, passing one of the members of the **D3DCMPFUNC** enumerated type to identify the desired comparison function.

Customizing the Stencil Buffer

[This is preliminary documentation and subject to change.]

Your application can customize the operation of the stencil buffer. It can set the comparison function, the stencil mask, and the stencil reference value. it can also control the action that Direct3D takes when the stencil test passes or fails. For more information, see Stencil Buffer State.

Vertex Buffers

[This is preliminary documentation and subject to change.]

This section introduces the concepts necessary to understand and use vertex buffers in a Direct3D application. Information is divided into the following sections:

- What Are Vertex Buffers?
- Vertex Buffer Descriptions
- Vertex Buffers and Device Types
- Using Vertex Buffers

What Are Vertex Buffers?

[This is preliminary documentation and subject to change.]

Vertex buffers, represented by the **IDirect3DVertexBuffer** interface, are simply memory buffers that contain vertex data. Vertex buffers can contain any vertex type — transformed or untransformed, lit or unlit — that can be rendered through the use of the vertex buffer rendering methods in the **IDirect3DDevice3** interface. You can process the vertices in a vertex buffer to perform operations such as transformation, lighting, or generating clipping flags. (Transformation is always performed.)

The flexibility of vertex buffers make them ideal staging points for reusing transformed geometry. You could create a single vertex buffer, transform, light, and clip the vertices in it, and render the model in the scene as many times as you need without retransforming it, even with interleaved render state changes. This can be very useful when rendering models that use multiple textures: the geometry is only transformed once, and then portions of it can be rendered as needed, interleaved with the required texture changes. Render state changes made after vertices are processed take effect the next time the vertices are processed. For more information, see Processing Vertices.

You can optimize geometry in vertex buffers to get maximum performance for vertex operations and rendering. See Optimizing a Vertex Buffer for details.

Note

Internally, vertex buffers use DirectDrawSurface objects for their memory management services. As a result, the semantics for accessing vertex buffer

memory are similar to those of DirectDrawSurface objects. In fact, the **IDirect3DVertexBuffer::Lock** method accepts the same flags as the **IDirectDrawSurface4::Lock** method. For more information, see Accessing Vertex Buffer Memory.

Vertex Buffer Descriptions

[This is preliminary documentation and subject to change.]

A vertex buffer is described in terms of its capabilities: if it can only exist in system memory, if it is only to be used for write operations, the type and number of vertices it can contain, and whether or not it has been optimized since it was created. All these traits are held within a **D3DVERTEXBUFFERDESC** structure.

Vertex buffer descriptions tell the system how to create a vertex buffer or tell your application how an existing buffer was created (and if it has been optimized since being created). You must specify a complete description to create a new vertex buffer, and you provide an empty description structure for the system to fill with the capabilities of a previously created vertex buffer. For more information about these tasks, see Creating a Vertex Buffer and Retrieving Vertex Buffer Descriptions.

The **dwSize** member, common to most DirectX structures, is used for identify structure versions and must be set to the structure size before the structure can be used with any methods. The **dwCaps** structure member contains general capability flags. The D3DVBCAPS_SYSTEMMEMORY flag indicates that the system should create (or already has created) the vertex buffer in system memory. Create an explicit system memory vertex buffer if your application uses a software rendering device; otherwise, it is best to let the system determine the best location by omitting the flag. For more information about explicit system memory vertex buffers, see Vertex Buffers and Device Types.

The presence of the D3DVBCAPS_WRITEONLY flag in **dwCaps** hints to the system that the vertex buffer memory is used only for write operations. This frees the driver to place the vertex data in the best possible memory location it can to enable fast processing and rendering. Reading from a vertex buffer that uses this flag can result in very slow memory access times. If the D3DVBCAPS_WRITEONLY flag isn't used, the driver is less likely to put the data in a location inefficient for read operations, sacrificing some processing and rendering speed. If no flags are specified, it is assumed that applications will perform read and write operations on the data within the vertex buffer.

Note

The D3DVBCAPS_OPTIMIZED flag is not used during vertex buffer creation. The system sets this capability when it optimizes a vertex buffer.

The final two **D3DVERTEXBUFFERDESC** structure members describe other capabilities. The **dwFVF** member contains a combination of flexible vertex format flags that identify the type of vertices that the buffer can contain. Vertex buffer

capacity is measured by the total number of vertices it can contain, given in the **dwNumVertices** member.

Vertex Buffers and Device Types

[This is preliminary documentation and subject to change.]

A Direct3D software rendering device can only be used with explicit system-memory vertex buffers. To create a vertex buffer in system memory, include the D3DVBDESC_SYSTEMMEMORY flag in the description structure that you provide to the **IDirect3D3::CreateVertexBuffer** method.

If your application uses a hardware accelerated device, it's best to omit the D3DVBDESC_SYSTEMMEMORY flag. This allows the system to place the vertex buffer in memory for best performance, but hardware devices can work with vertex buffers regardless of their location.

Using Vertex Buffers

[This is preliminary documentation and subject to change.]

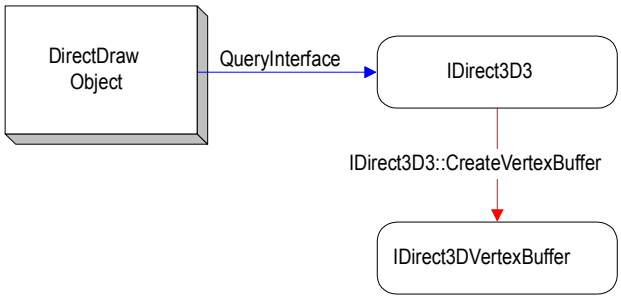
The following topics discuss common tasks that applications will perform when working with vertex buffers:

- Creating a Vertex Buffer
- Accessing Vertex Buffer Memory
- Processing Vertices
- Optimizing a Vertex Buffer
- Rendering From a Vertex Buffer
- Retrieving Vertex Buffer Descriptions

Creating a Vertex Buffer

[This is preliminary documentation and subject to change.]

The following figure illustrates the steps necessary to create a single vertex buffer.



You create a vertex buffer object by calling the **IDirect3D3::CreateVertexBuffer** method, which accepts four parameters. The first parameter is the address of a **D3DVERTEXBUFFERDESC** structure that describes the desired vertex format, buffer size, and general capabilities. These capabilities are detailed in Vertex Buffer Descriptions. Normally, the system automatically determines the best memory location (system or display memory) for the vertex buffer. However, software devices can only be used with explicit system-memory vertex buffers. For more information, see Vertex Buffers and Device Types.

The second parameter that **CreateVertexBuffer** accepts is the address of a variable that will be filled with a pointer to the new **IDirect3DVertexBuffer** interface of the vertex buffer object if the call succeeds. The third parameter determines if the vertex buffer will be capable of containing clipping information — in the form of clip flags — for vertices that exist outside the viewing area. Set this to 0 to create a "clipping-capable" vertex buffer, or include the D3DDP_DONOTCLIP flag to create a vertex buffer that cannot contain clip flags. The D3DDP_DONOTCLIP flag is only applied if you also indicate that the vertex buffer will contain transformed vertices (the D3DFVF_XYZRHW flag is included in the **dwFVF** member of the description structure). The **CreateVertexBuffer** method ignores the D3DDP_DONOTCLIP flag if you indicate that the buffer will contain untransformed vertices (the D3DFVF_XYZ flag). Clipping flags occupy additional memory, making a clipping-capable vertex buffer slightly larger than a vertex buffer incapable of containing clipping flags. Because these resources are allocated when the vertex buffer is created, you must request a clipping-capable vertex buffer ahead of time.

Note

Creating a vertex buffer that can contain clip flags does not necessarily mean that you must request that clip flags be generated during vertex processing or applied during rendering. Each vertex buffer rendering method accepts the D3DDP_DONOTCLIP flag to bypass clipping during rendering, and the **IDirect3DVertexBuffer::ProcessVertices** method accepts the D3DVOP_CLIP flag, which can be omitted to prevent the system from generating clip flags while it processes vertices.

There is no way to produce clip flags for a vertex buffer that was created without support for them. Attempts to use the **IDirect3DVertexBuffer::ProcessVertices** method to do this will fail in debug builds, returning D3DERR_INVALIDVERTEXFORMAT. Rendering methods will ignore clipping requests when rendering from a transformed vertex buffer that does not contain clip flags.

The last parameter that **CreateVertexBuffer** accepts is provided for future compatibility with COM aggregation features. Currently, aggregation isn't supported, so this parameter must be set to NULL.

The following example shows what creating a vertex buffer might look like in code:

```
/*
 * For the purposes of this example, the g_lpD3D variable is the
 * address of an IDirect3D3 interface exposed by a Direct3D
```

```
 * object, and the fIsAHardwareDevice variable is a BOOL variable
 * that is assumed to be set during application initialization.
 */

D3DVERTEXBUFFERDESC vbdesc;
ZeroMemory(&vbdesc, sizeof(D3DVERTEXBUFFERDESC));
vbdesc.dwSize        = sizeof(D3DVERTEXBUFFERDESC);
vbdesc.dwCaps        = 0L;
vbdesc.dwFVF         = D3DFVF_VERTEX;
vbdesc.dwNumVertices = NUM_FLAG_VERTICES;

// If this isn't a hardware device, make sure the
// vertex buffer uses system memory.
if( !fIsAHardwareDevice )
   vbdesc.dwCaps |= D3DVBCAPS_SYSTEMMEMORY;

// Create a clipping-capable vertex buffer.
if(FAILED(g_lpD3D->CreateVertexBuffer(&vbdesc,
                           &g_pvbVertexBuffer, 0L,
                           NULL)))
   return E_FAIL;
```

Accessing Vertex Buffer Memory

[This is preliminary documentation and subject to change.]

Vertex buffer objects enable applications to directly access the memory allocated for vertex data. You can retrieve a pointer to vertex buffer memory by calling the **IDirect3DVertexBuffer::Lock** method, then access the memory as needed to fill the buffer with new vertex data, or to read any data it already contains.

The **IDirect3DVertexBuffer::Lock** method accepts three parameters. The first, *dwFlags*, tells the system how the memory should be locked, and can be used to hint how the application will be accessing the data within the buffer. (You can hint for read-only or write-only access, which allows the driver to lock the memory to provide the best performance given the requested access type. These hints are not required, but can improve performance for memory access in some situations.) Because vertex buffers use DirectDrawSurface objects to contain vertex data, the flags that **IDirect3DVertexBuffer::Lock** accepts are identical to those accepted by the **IDirectDrawSurface4::Lock** method, with identical results.

The second parameter accepted by the **Lock** method, *lplpData*, is the address of an **LPVOID** variable that will contain a valid pointer to the vertex buffer memory if the call succeeds. The last parameter, *lpdwSize*, is the address of a variable that will contain the size, measured in bytes, of the buffer at *lplpData* after the call returns. You can set *lpdwSize* to NULL if your application doesn't need information about the buffer size.

Performance Notes

Use the DDLOCK_READONLY flag if your application will only be reading from the vertex buffer memory. Including this flag enables Direct3D to optimize its internal procedures to improve efficiency, given that access to the memory will be read-only. Although it is possible to write to memory locked with the DDLOCK_READONLY flag, doing so can produce unexpected results. In addition, attempting to read from a vertex buffer that was created with the D3DVBCAPS_WRITEONLY flag can be extremely slow, even if you lock the buffer for read-only access.

The vertex buffer memory itself is a simple array of vertices, specified in flexible vertex format. If your application is uses the legacy vertex structures, **D3DVERTEX**, **D3DLVERTEX**, and **D3DTLVERTEX**, the stride is simply the size of the structure, in bytes. If you are using a vertex format different from the legacy formats, use the stride of whatever vertex format structure you define. You can calculate the stride of each vertex at run time by examining the flexible vertex format flags contained within the vertex buffer description. The following table shows the size for each vertex component.

Vertex Format Flag	Size
D3DFVF_DIFFUSE	**sizeof(DWORD)**
D3DFVF_NORMAL	**sizeof(float)** × 3
D3DFVF_SPECULAR	**sizeof(DWORD)**
D3DFVF_TEX*n*	**sizeof(float)** × 2 × *n*
D3DFVF_XYZ	**sizeof(float)** × 3
D3DFVF_XYZRHW	**sizeof(float)** × 4

The number of texture coordinates present in the vertex format is described by the D3DFVF_TEX*n* flags (where *n* is a value from 0 to 8). Because each set of texture coordinates in the vertex format occupies the space of two **float** variables, multiply the number of texture coordinate sets by the size of one set of texture coordinates to calculate the memory required for that number of texture coordinates.

Use the total vertex stride to increment and decrement the memory pointer as needed to access particular vertices.

Processing Vertices

[This is preliminary documentation and subject to change.]

Processing the vertices in a vertex buffer applies the current transformation matrices for the device, and can optionally apply vertex operations such as lighting, generating clip flags, and updating extents. The **IDirect3DVertexBuffer** interface exposes the **IDirect3DVertexBuffer::ProcessVertices** method to process vertices.

You process vertices from a source vertex buffer into a destination vertex buffer by calling the **ProcessVertices** method of the destination vertex buffer, not the source buffer. The method accepts seven parameters that describe the operations to be

performed, the location of the source vertex buffer's **IDirect3DVertexBuffer** interface, the rendering device that will perform the vertex operations, and the location and quantity of vertices that the method targets. After the call, the destination buffer contains the processed vertex data, and the source buffer is unchanged.

When preparing to process vertices, set the first parameter, *dwVertexOp*, to indicate the vertex operations you want to perform. You must include the D3DVOP_TRANSFORM flag, or the method will fail, but the remaining operations are optional. You can include any combination of optional flags to light the vertices, generate clip flags, and update extents during vertex processing.

The second and third parameters, *dwDestIndex* and *dwCount*, reflect the index within the destination buffer at which the vertices will be placed and the total number of vertices that should be processed and placed in the destination buffer. The fourth parameter, *lpSrcBuffer,* should be set to the address of the **IDirect3DVertexBuffer** of the vertex buffer object that contains the source vertices. The *dwSrcIndex* specifies the index at which the method should start processing vertices. (The total number of source vertices to be processed is implied from the *dwCount* parameter.) Set the *lpD3DDevice* parameter to the address of the **IDirect3DDevice3** interface for the rendering device that process the vertices. The final parameter is reserved for future use, and must be set to 0.

Take care to create vertex buffers that use compatible vertex formats. At the least, the source buffer should contain untransformed vertices (using the D3DFVF_XYZ flag in the vertex format of the buffer description), and the destination buffer should contain transformed vertices (using the D3DFVF_XYZRHW flag). Any lighting or clipping services require that the source and destination vertex formats contain the appropriate fields. For instance, don't request lighting on vertices when the vertex format doesn't include a vertex normal. Likewise, you can't request that the system produce clip flags for a destination vertex buffer that was created without clipping capabilities. Attempts to perform operations on incompatible buffers will fail in debug builds.

You cannot process vertices when the source or destination vertex buffers are locked.

Optimizing a Vertex Buffer

[This is preliminary documentation and subject to change.]

Optimizing a vertex buffer causes the system to modify the contents of the buffer for better performance when processing or rendering vertices. Exactly how vertices are optimized is platform-specific and subject to change. Consequently, you cannot lock or otherwise access the contents of an optimized vertex buffer.

Call the **IDirect3DVertexBuffer::Optimize** method to optimize a vertex buffer. Optimizing a vertex buffer improves performance of vertex operations and rendering. The **IDirect3DVertexBuffer::Optimize** method accepts two parameters, only one of which is used. Set the *lpD3DDevice* parameter to the address of the device for which the vertices will be optimized, and set the last parameter to 0. Locked vertex buffers cannot be optimized until they are unlocked.

You can improve performance by keeping vertices in optimized format. However, optimized vertex buffers should be used only for static geometry, because once a vertex buffer is optimized, you can't lock it to change the optimized contents. After you optimize a vertex buffer, it can only be used with the **IDirect3DVertexBuffer::ProcessVertices** method and the vertex buffer rendering methods.

Rendering From a Vertex Buffer

[This is preliminary documentation and subject to change.]

As introduced in Rendering, the **IDirect3DDevice3** interface includes methods to render primitives from a vertex buffer. These methods work very much like other rendering methods in **IDirect3DDevice3**, but employ slightly different parameters to accommodate vertex buffer objects.

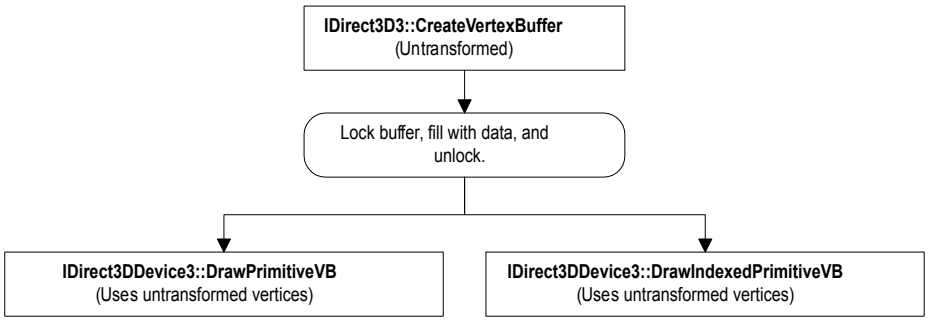
The following topics introduce common rendering situations, and discuss specific issues related to calling the vertex buffer rendering methods:

- About Vertex Buffer Rendering
- Calling Vertex Buffer Rendering Methods

About Vertex Buffer Rendering

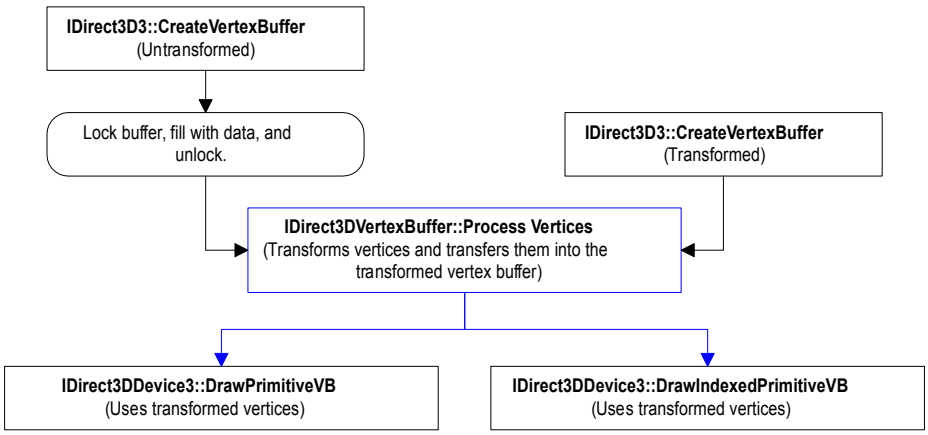
[This is preliminary documentation and subject to change.]

There are two common situations in which your application will render vertices from a vertex buffer. At the most basic level, the two situations break down according to the type of vertex that is in the vertex buffer at "render time," but indirectly, the procedure your application uses also determines when vertex and rendering operations occur. The following diagram illustrates the process of rendering from an untransformed vertex buffer.



As shown in the preceding illustration, the **IDirect3DDevice3::DrawPrimitiveVB** and **IDirect3DDevice3::DrawIndexedPrimitiveVB** methods are capable of rendering from a non-transformed vertex buffer. In this case, the system performs vertex and rendering operations each time you call a rendering method. For DirectX 6.0, using this approach isn't likely to provide improved performance over traditional DrawPrimitive rendering methods, but it might be more convenient in some

situations. You can optimize performance by reusing transformed vertex data when you can, as shown in the following illustration.



In this case, your application creates two vertex buffers: one for untransformed geometry, and another for transformed geometry. The second buffer receives transformed vertex data when the **IDirect3DVertexBuffer::ProcessVertices** is called. The **ProcessVertices** method reads the vertices in the source buffer, performs the requested vertex operations on them, and places the results in the destination buffer. You can call the same rendering methods for transformed vertices that you would to render untransformed vertices. However, unlike untransformed vertices, Direct3D automatically detects that the data in the vertex buffer is transformed, sending it be rasterized right away. The performance overhead is kept to a minimum by eliminating unnecessary transformations.

You can optimize vertex buffer contents to increase performance even more. For more information, see Optimizing a Vertex Buffer.

Calling Vertex Buffer Rendering Methods

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::DrawPrimitiveVB** method corresponds to the **IDirect3DDevice3::DrawPrimitive** method. Like its relative, the **DrawPrimitiveVB** method assumes that vertices appear in sequential order within the vertex buffer.

The **DrawPrimitiveVB** method accepts five parameters. Set the *d3dptPrimitiveType* parameter to indicate the type of primitive being rendered by using one the members of the **D3DPRIMITIVETYPE** enumerated type. Specify the address of the vertex buffer that contains the vertices in the second parameter, *lpd3dVertexBuffer*, then set values in the *dwStartVertex* and *dwNumVertices* parameters to reflect the first vertex and the total number of vertices that will be rendered. You don't need to render all the vertices that the vertex buffer contains, but you must use the appropriate number of vertices for that primitive type (as documented in the reference information for **D3DPRIMITIVETYPE**). The last parameter, *dwFlags*, determines the rendering behavior. These flags are identical to those used with other rendering methods. You

can set flags to enable lighting and clipping, or to update the viewport extents during rendering.

The **IDirect3DDevice3::DrawIndexedPrimitiveVB** method renders primitives by indexing the vertices within a vertex buffer. The first two parameters are identical to those of the **DrawPrimitiveVB** method. The third parameter, *lpwIndices*, should be set to the address of an ordered array of **WORD** indices that the method will use to access the vertices to be rendered, and the fourth parameter, *dwIndexCount*, should be set to the number of index values that the array contains. The *dwFlags* parameter is identical to its counterpart in the **DrawPrimitiveVB** method.

You cannot request that vertex buffer rendering methods perform lighting if the format of the vertices within the buffer does not contain a vertex normal. Nor can you request that the rendering methods clip vertices when rendering from a transformed vertex buffer created with the D3DDP_DONOTCLIP flag. (This is allowed for untransformed vertex buffers, as the system will create a clipping-capable version at render time.) Attempts to request unavailable services will not cause the method to fail in debug builds, but the services will not be applied.

A software device, as opposed to a hardware-accelerated device, can only render from vertex buffers that were created with the D3DVBCAPS_SYSTEMMEMORY flag. For more information, see Vertex Buffers and Device Types.

Retrieving Vertex Buffer Descriptions

[This is preliminary documentation and subject to change.]

Applications often need to retrieve information about existing vertex buffers at run time. Direct3D makes this possible through the **IDirect3DVertexBuffer::GetVertexBufferDesc** method. Call this method using the **IDirect3DVertexBuffer** interface of any vertex buffer to retrieve its current description.

The **GetVertexBufferDesc** accepts only one parameter: the address of a properly initialized **D3DVERTEXBUFFERDESC** structure. To initialize the structure, set the **dwSize** member to the structure size, in bytes, and set the remaining members to 0. After the method returns, the structure will contain information about the vertex buffer capabilities, the vertex format in the buffer, and the total number of vertices that it can contain.

Note

Optimizing a vertex buffer sets the D3DVBCAPS_OPTIMIZED capability flag in the **dwFlags** member of the **D3DVERTEXBUFFERDESC** structure. If this flag is set, the vertex buffer cannot be locked, and its contents are not made available to anything but rendering methods and the **IDirect3DVertexBuffer::ProcessVertices** method.

For general information, see Vertex Buffer Descriptions.

Common Techniques and Special Effects

[This is preliminary documentation and subject to change.]

Direct3D provides a powerful set of tools that can be used to increase the realistic appearance of a 3-D scene. This section presents information on some common special effects that can be produced with Direct3D. The range of possible effects is by no means limited to those presented here. The discussion in this section is organized into the following topics:

- Fog
- Billboarding
- Clouds, Smoke, and Vapor Trails
- Texture Blending Techniques
- Fire, Flares, Explosions, and More
- Motion Blur
- Stencil Buffer Techniques
- Colored Lights
- Antialiasing

Fog

[This is preliminary documentation and subject to change.]

The following topics introduce fog and present information about using various fog features in Direct3D applications:

- Introduction to Fog
- Fog Formulas
- Fog Blending
- Fog Color
- Range-Based Fog
- Pixel Fog
- Vertex Fog

Introduction to Fog

[This is preliminary documentation and subject to change.]

Adding fog to a 3-D scene can enhance realism, provide ambience or set a mood, and obscure artifacts sometimes caused when distant geometry comes into view. Direct3D Immediate Mode supports two fog models, pixel fog and vertex fog, each with its own features and programming interface.

Essentially, fog is implemented by blending the color of objects in a scene with a chosen fog color based on the depth of an object in a scene, or its distance from the viewpoint. As objects grow more distant, their original color blends more and more into the chosen fog color, creating the illusion that the object is being increasingly obscured by tiny particles floating in the scene — fog. The following screen capture shows a scene rendered without fog, and a similar scene rendered with fog enabled.

In the preceding figure, the scene on the left has a clear horizon, beyond which no scenery is visible, even though it would be visible in the real world. The scene on the right obscures the horizon by using a fog color identical to the background color, making polygons appear to fade into the distance. By combining discrete fog effects with creative scene design you can add mood and soften the color of objects in a scene.

Direct3D provides applications with two ways to add fog to a scene, pixel fog and vertex fog, named according to how the fog effects are applied. For details, see Pixel Fog and Vertex Fog.

Fog Formulas

[This is preliminary documentation and subject to change.]

Applications can control how fog affects the color of objects in a scene by changing how Direct3D computes fog effects over distance. The **D3DFOGMODE** enumerated type contains members that identify the three fog formulas. All formulas calculate a fog factor as a function of distance, given parameters that your application sets. How distance itself is computed varies on the projection matrix or if range-based fog is enabled. For more information, see Eye-Relative vs. Z-Based Depth and Range-Based Fog.

D3DFOG_LINEAR

$$f = \frac{end - d}{end - start}$$

In the linear formula, *start* is the distance at which fog effects begin, *end* is the distance at which fog effects no longer increase, and *d* represents depth (or distance from the viewpoint) within a scene. Values for *d* increase as objects become more distant. The linear formula is supported for both pixel fog and vertex fog, but the exponential formulas are currently only supported when using pixel fog:

D3DFOG_EXP

$$f = 1 / e^{d \times density}$$

D3DFOG_EXP2

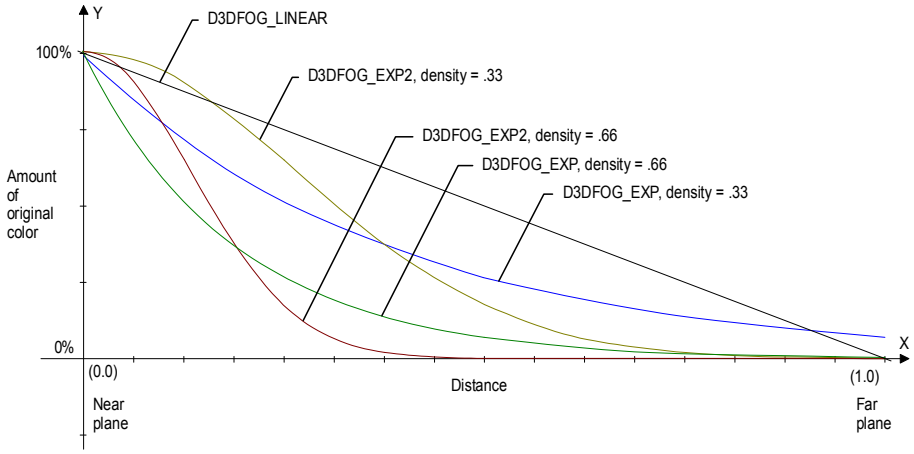
$$f = 1 / e^{(d \times density)^2}$$

In the preceding two exponential formulas, *e* is the base of natural logarithms (approximately 2.71828), *density* is an arbitrary fog density that can range from 0.0 to 1.0, and *d* is depth (or distance from the viewpoint) within a scene.

Note

The system stores the fog factor in the alpha component of the specular color for a vertex. If your application performs its own transformation and lighting, you can insert fog factor values manually, to be applied by the system during rendering.

The following illustration graphs these formulas, using common values as in the formula parameters.



When Direct3D calculates fog effects, it uses the fog factor from one of the preceding equations in a blending formula, shown here:

$$C = f \cdot C_i + (1 - f) \cdot C_f$$

This formula effectively scales the color of the current polygon C_i by the fog factor *f*, and adds the product to the fog color C_f scaled by the bitwise inverse of the fog factor. The resulting color value is a blend of the fog color and the original color, as a factor of distance. The formula applies to all devices supported in DirectX 6.0. For the legacy ramp device, the fog factor scales the diffuse and specular color components, clamped to the range of 0.0 and 1.0, inclusive. The fog factor typically starts at 1.0 for the near plane and decreases to 0.0 at the far plane.

Fog Blending

[This is preliminary documentation and subject to change.]

Fog blending refers to the application of the fog factor to the fog and object colors to produce the final color that appears in a scene, as discussed in the Fog Formulas topic. The D3DRENDERSTATE_FOGENABLE render state controls fog blending.

Set this render state to TRUE to enable fog blending (the default is FALSE), as in the following example code:

```
//
// For this example, g_lpDevice is a valid pointer
// to an IDirect3DDevice3 interface.
HRESULT hr;
hr = g_lpDevice->SetRenderState(
                D3DRENDERSTATE_FOGENABLE,
                TRUE);
if FAILED(hr)
   return hr;
```

You must enable fog blending for both pixel fog and vertex fog. For information about using these types of fog, see Pixel Fog and Vertex Fog.

Fog Color

[This is preliminary documentation and subject to change.]

Fog color for both pixel and vertex fog is set through the D3DRENDERSTATE_FOGCOLOR render state. The render state values can be any RGB color, specified as an RGBA color (the alpha component is ignored).

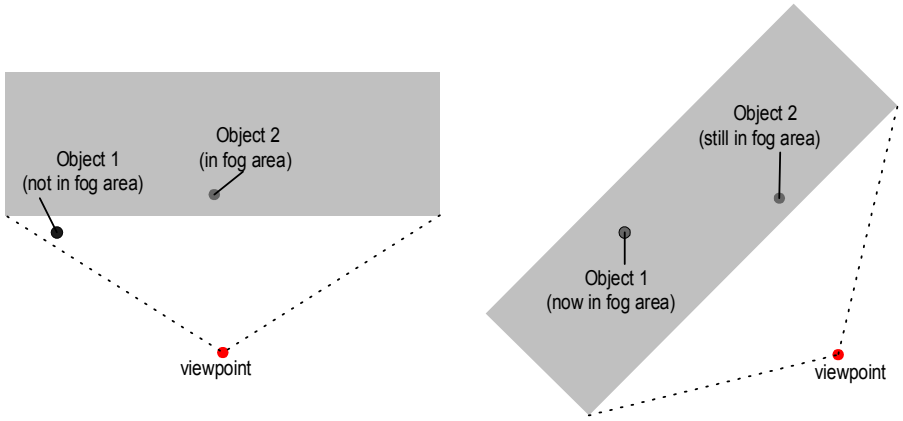
The following example sets the fog color to white:

```
/* For this example, the g_lpD3DDevice variable is
 * a valid pointer to an IDirect3DDevice3 interface.
 */
HRESULT hr;

hr = pd3dDevice->SetRenderState(
                D3DRENDERSTATE_FOGCOLOR,
                0x00FFFFFF); // Highest 8 bits aren't used.

if(FAILED(hr))
   return hr;
```

Range-Based Fog

[This is preliminary documentation and subject to change.]

Sometimes, using fog can introduce graphic artifacts that cause objects to be blended with the fog color in non-intuitive ways. For example, imagine a scene in which there are two visible objects, one distant enough to be affected by fog, and the other near enough to be unaffected. If the viewing area rotates in place, the apparent fog effects can change, even if the objects are stationary. The following illustration shows a top-down view of such a situation.



Range-based fog is another, more accurate, way to determine the fog effects. In range-based fog, Direct3D uses the actual distance from the viewpoint to a vertex for its fog calculations, increasing the effects of fog as the distance between the two points increases, rather than the depth of the vertex within in the scene, thereby avoiding rotational artifacts.

If the current device supports range-based fog, it will set the D3DPRASTERCAPS_FOGRANGE capability flag in the **dwRasterCaps** member of the **D3DPRIMCAPS** structure when you call the **IDirect3DDevice3::GetCaps** method. To enable range-based fog, set the D3DRENDERSTATE_RANGEFOGENABLE render state to TRUE.

Range-based fog is computed by Direct3D during transformation and lighting. As discussed in About Vertex Fog, applications that don't use the Direct3D transformation and lighting engine must also perform their own vertex fog calculations. In this case, provide the range-based fog factor in the alpha component of the specular component for each vertex.

Note

No hardware currently exists that supports per-pixel range-based fog. As a result, Direct3D performs range-based fog only when using vertex fog with the Direct3D transformation and lighting engine. If your application performs its own transformation and lighting it must perform its own fog calculations, range-based or otherwise.

Pixel Fog

[This is preliminary documentation and subject to change.]

This section introduces the concept of pixel fog and provides information about using it in Direct3D applications. Information is divided into the following topics:

- About Pixel Fog
- Eye-Relative vs. Z-Based Depth

- Pixel Fog Parameters
- Using Pixel Fog

About Pixel Fog

[This is preliminary documentation and subject to change.]

Pixel fog gets its name from the fact that it is calculated on a per-pixel basis in the device driver. (This is unlike vertex fog, in which Direct3D computes fog effects when it performs transformation and lighting.) Pixel fog is sometimes called "table fog" because some drivers use a precalculated look-up table to determine the fog factor (using the depth of each pixel) to be applied in blending computations. Pixel fog can be applied using any of the fog formulas identified by members of the **D3DFOGMODE** enumerated type. Pixel-fog formula implementations are driver-specific, and if a driver doesn't support a complex fog formula, it should degrade to a less complex formula.

Note

As discussed in Range-Based Fog, pixel fog does not support range-based fog calculations.

Eye-Relative vs. Z-based Depth

[This is preliminary documentation and subject to change.]

To alleviate fog-related graphic artifacts caused by uneven distribution of z-values within a depth buffer, most hardware devices use eye-relative depth instead of z-based depth values for pixel fog. Eye-relative depth is essentially the w element from a homogenous coordinate set. (Direct3D takes the reciprocal of the RHW element from a device space coordinate set to reproduce true w.) If a device supports eye-relative fog, it sets the D3DPRASTERCAPS_WFOG flag in the **dwRasterCaps** member of the **D3DPRIMCAPS** when you call **IDirect3DDevice3::GetCaps** method. (The **D3DDEVICEDESC** structure you pass to **GetCaps** contains multiple **D3DPRIMCAPS** structures that describe capabilities for various types of primitives.)

Note

With the exception of the reference rasterizer, software devices always use z to calculate pixel fog effects.

When eye-relative fog is supported, the system will automatically use eye-relative depth in favor of z-based depth if the provided projection matrix produces z-values in world space that are equivalent to w-values in device space. (You set the projection matrix by calling the **IDirect3DDevice3::SetTransform** method, using the D3DTRANSFORMSTATE_PROJECTION value and passing a **D3DMATRIX** structure that represents the desired matrix.) If the projection matrix isn't compliant with this requirement, fog effects will not be applied properly. For details about producing a compliant matrix, see A W-Friendly Projection Matrix. (The perspective projection matrix provided in What Is the Projection Transformation? produces a compliant projection matrix.)

Usage Notes

To use eye-relative fog, perspective-correct texture mapping must be enabled. To enable perspective-correct texture mapping, set the D3DRENDERSTATE_TEXTUREPERSPECTIVE render state to TRUE. For DirectX 6.0 and later, this is the default value.

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

Direct3D checks the fourth column of the projection matrix, and if the coefficients are [0,0,0,1] (for an affine projection) the system will use z-based depth values for fog. In this case, you must also specify the start and end distances for linear fog effects in device space, which ranges from 0.0 at the nearest point to the user, and 1.0 at the farthest point.

Pixel Fog Parameters

[This is preliminary documentation and subject to change.]

All parameters for pixel fog are controlled through device render states. Pixel fog supports all of the formulas introduced in Fog Formulas. Choose the formula you want the system to use by setting the D3DRENDERSTATE_FOGTABLEMODE to the desired member from the **D3DFOGMODE** enumerated type.

When using the linear fog formula, you set the starting and ending distances through the D3DRENDERSTATE_FOGTABLESTART and D3DRENDERSTATE_FOGTABLEEND render states. These distances are in world-space units except when using a software device, in which case distances are in device space, which ranges from 0.0 at the nearest visible point to 1.0 at the farthest visible point.

The D3DRENDERSTATE_FOGTABLEDENSITY render state controls the fog density applied when an exponential fog formula is enabled. Fog density is a essentially weighting factor, ranging from 0.0 to 1.0 (inclusive), that scales the distance value in the exponent.

As discussed in Fog Color, the D3DRENDERSTATE_FOGCOLOR render state controls what color the system uses for fog blending.

Using Pixel Fog

[This is preliminary documentation and subject to change.]

Use the following steps to enable pixel fog in your application:

Û To enable pixel fog

1. Enable fog blending by setting the D3DRENDERSTATE_FOGENABLE render state to TRUE.
2. Set the desired fog color in the D3DRENDERSTATE_FOGCOLOR render state.

3. Choose the fog formula you want to use by setting the D3DRENDERSTATE_FOGTABLEMODE render state to the corresponding member of the **D3DFOGMODE** enumerated type.
4. Set the fog parameters as desired for the selected fog mode in the associated render states. This includes the start and end distances for linear fog, and fog density for exponential fog mode.

The following example shows what these steps might look like in code:

```
// For brevity, error values in this example are not checked
// after each call. A real-world application should check
// these values appropriately.
//
// For the purposes of this example, g_lpDevice is a valid
// pointer to an IDirect3DDevice3 interface.
void SetupPixelFog(DWORD dwColor, DWORD dwMode)
{
   float fStart = 0.5f,    // for linear mode
         fEnd   = 0.8f,
         fDensity = 0.66;  // for exponential modes

   // Enable fog blending.
   g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGENABLE, TRUE);

   // Set the fog color.
   g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGCOLOR, dwColor);

   // Set fog parameters.
   if(D3DFOG_LINEAR == dwMode)
   {
      g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGTABLEMODE, dwMode);
      g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGTABLESTART, *(DWORD *)
(&fStart));
      g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGTABLEEND,   *(DWORD *)
(&fEnd));
   }
   else
   {
      g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGTABLEMODE, dwMode);
      g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGTABLEDENSITY, *(DWORD *)
(&fDensity));
   }

}
```

Note

Some fog parameters are required as floating-point values, even though the **IDirect3DDevice3::SetRenderState** method only accepts **DWORD** values in the second parameter. The preceding example provides the floating-point values to **SetRenderState** without data translation by casting the addresses of the floating-point variables as **DWORD** pointers, then dereferencing them.

Vertex Fog

[This is preliminary documentation and subject to change.]

This section introduces vertex fog and provides details about using it in Direct3D applications. Information is divided into the following topics:

- About Vertex Fog
- Vertex Fog Parameters
- Using Vertex Fog

About Vertex Fog

[This is preliminary documentation and subject to change.]

When the system performs vertex fogging, it applies fog calculations at each vertex in a polygon, then interpolates the results across the face of the polygon during rasterization. Unlike pixel fog, vertex fog only supports the linear fog formula (D3DFOG_LINEAR). Because vertex fog effects are computed by the Direct3D lighting and transformation engine, most vertex fog parameters are exposed in the device lighting states set through the **IDirect3DDevice3::SetLightState** method. For more information, see Vertex Fog Parameters.

If your applications doesn't use Direct3D for transformation and lighting, it must perform fog calculations on its own. In this case, your application can place the fog factor it computes in the alpha component of the specular color for each vertex. You are free to use whatever formulas you desire — range-based or otherwise. Direct3D uses the supplied fog factor to interpolate across the face of each polygon. Applications that do not use Direct3D transformation and lighting need not set vertex fog parameters through light states, but must still enable fog and set the fog color through the associated render states. For more information, see Vertex Fog Parameters.

Vertex Fog Parameters

[This is preliminary documentation and subject to change.]

Vertex fog parameters related to the chosen fog formula are controlled by setting specific device lighting states (not render states) with the **IDirect3DDevice3::SetLightState** method. Currently, the only supported formula for vertex fog is the linear formula, which you can choose by setting the D3DLIGHTSTATE_FOGMODE lighting state to D3DFOG_LINEAR. Set the starting and ending distances for linear fog through the

D3DLIGHTSTATE_FOGSTART and D3DLIGHTSTATE_FOGEND lighting states. All distances are in world space.

Note

Although exponential fog formulas are not currently supported for vertex fog, the **D3DLIGHTSTATETYPE** enumerated type used with the **SetLightState** method contains a D3DLIGHTSTATE_FOGDENSITY member. This member is not currently used, and is included to allow for future expansion.

The color that the system uses for fog blending is controlled through the D3DRENDERSTATE_FOGCOLOR device render state. For more information, see Fog Color and Fog Blending.

Applications that perform their own transformation and lighting must also perform their own vertex fog calculations, and as such they probably wouldn't use light states at all. As a result, such an application need only enable fog blending and set the fog color through the associated render states, as described in Fog Blending and Fog Color.

Using Vertex Fog

[This is preliminary documentation and subject to change.]

Use the following steps to enable vertex fog in your application:

Ô To enable vertex fog

1. Enable fog blending by setting D3DRENDERSTATE_FOGENABLE to TRUE.
2. Set the fog color in the D3DRENDERSTATE_FOGCOLOR render state.
3. Choose the desired fog formula by setting the D3DLIGHTSTATE_FOGMODE lighting state to a member of the **D3DFOGMODE** enumerated type. (Currently, only D3DFOG_LINEAR is supported for vertex fog.)
4. Set the fog parameters as desired for the selected fog formula in the associated lighting states. Do not use the fog parameter related render states — those are used only for pixel fog.

The following example shows what these steps might look like in code:

```
// For brevity, error values in this example are not checked
// after each call. A real-world application should check
// these values appropriately.
//
// For the purposes of this example, g_lpDevice is a valid
// pointer to an IDirect3DDevice3 interface.
void SetupVertexFog(DWORD dwColor, BOOL fUseRange)
{
    float fStart = 0.5f,    // linear fog distances
          fEnd   = 0.8f;
```

```
    // Enable fog blending.
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGENABLE, TRUE);

    // Set the fog color.
    g_lpDevice->SetRenderState(D3DRENDERSTATE_FOGCOLOR, dwColor);

    // Set fog parameters.
    //
    // REMEMBER: Vertex fog formula parameters are connected
    // to the lighting states, not render states.
    g_lpDevice->SetLightState(D3DLIGHTSTATE_FOGMODE, D3DFOG_LINEAR);
    g_lpDevice->SetLightState(D3DLIGHTSTATE_FOGSTART, *(DWORD *)(&fStart));
    g_lpDevice->SetLightState(D3DLIGHTSTATE_FOGEND,   *(DWORD *)(&fEnd));

    // Enable range-based fog if desired (only supported for vertex fog).
    // For this example, it is assumed that fUseRange is set to non-zero
    // only if the driver exposes the D3DPRASTERCAPS_FOGRANGE capability.
    //
    // Note: this is slightly more performance intensive
    //       than non-range-based fog.
    if(fUseRange)
        g_lpDevice->SetRenderState(
                    D3DRENDERSTATE_RANGEFOGENABLE,
                    TRUE);
}
```

Note

Some fog parameters are required as floating-point values, even though the **IDirect3DDevice3::SetRenderState** and **IDirect3DDevice3::SetLightState** methods only accept **DWORD** values in the second parameter. The preceding example successfully provides the floating-point values to these methods without data translation by casting the addresses of the floating-point variables as **DWORD** pointers, then dereferencing them.

Billboarding

[This is preliminary documentation and subject to change.]

When creating 3-D scenes, applications can sometimes gain performance advantages by rendering 2-D objects in a way that makes them appear to be 3-D objects. This is the basic idea behind the technique of billboarding.

A billboard in the normal sense of the word is a sign along a roadway. Direct3D applications can easily create and render this type of billboard by defining a rectangular solid and applying a texture to it. Billboarding in the more specialized sense of 3-D graphics is an extension of this. The goal is to make 2-D objects appear

to be 3-D. The technique is to apply a texture containing the object's image to a rectangular primitive. The primitive is rotated so that it always faces the viewer. It doesn't matter if the object's image is not rectangular. Portions of the billboard can be made transparent, so the parts of the billboard image that you don't want seen are not visible.

Many games employ billboarding for their animated sprites. For instance, when the player is moving through a 3-D maze, he or she may see weapons or rewards that can be picked up. These are typically 2-D images textured onto a rectangular primitive. Billboarding is often used in games to render images of trees and bushes.

When an image is applied to a billboard, the rectangular primitive must first be rotated so that the resulting image faces the viewer. Your application must then translate it into position. The program can then apply a texture to the primitive.

Billboarding works best for symmetrical objects, especially objects that are symmetrical around the vertical axis. It also requires that the altitude of the viewpoint doesn't increase too much. If the user is allowed to view the billboarded from above, it will become readily apparent that the object is 2-D rather than 3-D.

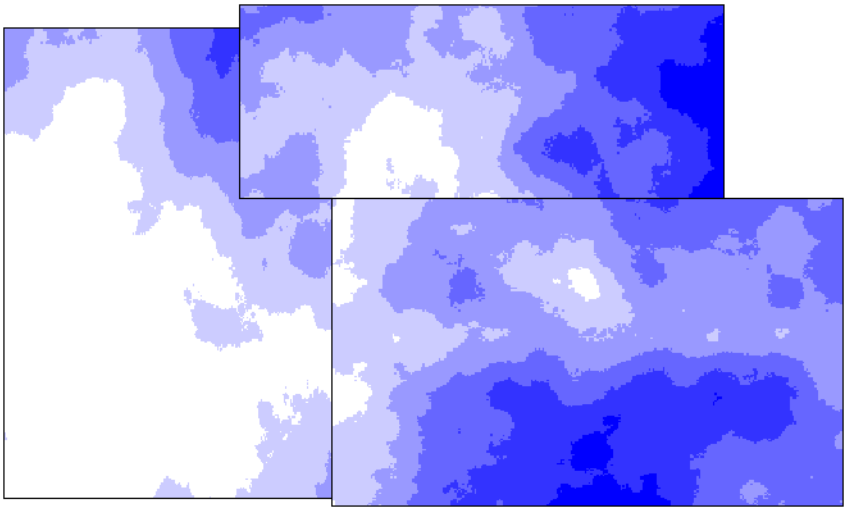
Clouds, Smoke, and Vapor Trails

[This is preliminary documentation and subject to change.]

Clouds, smoke, and vapor trails can all be created by an extension of the billboarding technique (see Billboarding). By rotating the billboard around two axes instead of one, your application can enable the viewer to look at a billboard from any angle. Typically, your application will rotate the billboard around the horizontal and vertical axes.

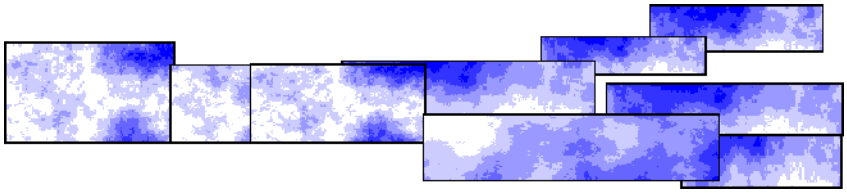
To make a simple cloud, your application can rotate a rectangular primitive around one or two axes so that it faces the viewer. A cloud-like texture can then be applied to the primitive with transparency. For details on applying transparent textures to primitives, see Texture Blending. The cloud can be animated by applying a series of textures over time.

Applications can created more complex clouds by forming them from a group of primitives. The each part of the cloud is a rectangular primitive. The primitives can be moved independently over time to give the appearance of a dynamic mist. This concept is illustrated in the following figure.



The appearance of smoke is displayed in a manner similar to clouds. It typically requires multiple billboards, like complex clouds. Smoke usually billows and rises over time, so the billboards that make up the smoke plume need to be moved accordingly. More billboards may need to be added as the plume rises and disperses.

A vapor trail is just a smoke plume that doesn't rise. However, like a smoke plume, it does disperse over time. The figure below illustrates the technique of using billboards to simulate a vapor trail.



Texture Blending Techniques

[This is preliminary documentation and subject to change.]

The power of texture blending as tool for producing realism in a 3-D scene goes beyond adding finishes and light maps to primitives. In fact, it is fair to say that not even cutting-edge researchers have yet discovered all of the possibilities that the current texture blending features offer. There are, however, common special effects that are widely used in 3-D applications. Direct3D applications provides copious support these techniques.

This section provides an overview of common uses of texture blending to produce special effects in a 3-D scene or enhance its appearance of realism. The information is divided into the following topics:

- Bump Mapping

- Detail Texture Mapping

Bump Mapping

[This is preliminary documentation and subject to change.]

Texture blending enables applications to create the appearance of a complex texture on a primitive. This works well for smooth surfaces. For instance, your application can use the texture blending feature of Direct3D to apply a smooth wood grain finish onto a tabletop in a scene.

However, texture blending alone is insufficient to model rough surfaces such as the bark of a tree. Fortunately, the Direct3D enables applications to easily use a technique called bump mapping. A Bump map is a texture that stores depth information. That is, it stores the values indicating the high and low spots on a surface.

Your application applies the bump map to the texture using blending stages. Set the texture blending operation of the blending stage containing the bump map to D3DTOP_BUMPENVMAP or D3DTOP_BUMPENVMAPLUMINANCE. For more information, see D3DTEXTUREOP and Creating Blending Stages.

Detail Texture Mapping

[This is preliminary documentation and subject to change.]

Through multitexturing or multipass rendering, Direct3D enables your application to apply detail textures to primitives. Use detail textures to apply scuff marks, bumps, and other surface attributes that give the appearance of realism. You may want to use mipmaps for applying detail textures. See Texture Filtering With Mipmaps.

Detail textures can also be used as depth cues. For instance, suppose your application simulates landing a helicopter. As the helicopter approaches the ground, it is not unusual for the ground textures to be magnified so much that they begin to look fuzzy. In this situation, the user would have difficulty distinguishing the distance to the ground. Adding a detail texture that resembles gravel as the user approaches the ground helps the user derive sufficient depth cues to pilot the helicopter properly.

If the viewer is far from the primitive, you probably don't want the details shown. Be aware, that a primitive may appear brighter when your application does not apply the detail texture. One way to compensate is by applying a light map texture to darken the primitive.

Fire, Flares, Explosions, and More

[This is preliminary documentation and subject to change.]

Applications can use Direct3D to simulate natural phenomena involving energy releases. For instance, a program can generate the appearance of fire by applying flame-like textures to a set of billboards. This is especially effective if the program uses a sequence of fire textures to animate the flames on each billboard in the fire.

Varying the speed of the animation playback from billboard to billboard increases the appearance of real flames. The semblance of intermingled 3-D flames can be achieved by layering the billboards, and the textures on the billboards.

Flares and flashes can be simulated by applying successively brighter light maps to all primitives in a scene. Although this is a computationally high-overhead technique, it allows your program to simulate a localized flare or flash. That is, the portion of the scene where the flare or flash originates can be brightened first.

Another technique is to position a billboard in front of the viewport so that the entire viewport is covered. The program applies successively whiter textures to the billboard and decreases the transparency over time. The entire scene will fade to white as time passes. This is a low overhead method of creating a flare. However, using this technique, it can be difficult to generate the appearance of a bright flash from a single point light source.

Explosions can be displayed in a 3-D scene procedures like those used for fire, flashes, and flares. For instance, your program might use a billboard to display a shock wave an rising plume of smoke when the explosion occurs. At the same time, your application can use a set of billboards to simulate flames. In addition, it can position a single billboard in front of the viewport to add a flare of light to the entire scene.

Energy beams can be simulated using billboards. Your application can also display them using primitives that are defined as line lists or line strips. For details, see Line Lists and Line Strips.

Your application can create force fields using billboards or primitives defined as triangle lists. To create a force field from triangle lists, define a set of disjoint triangles in a triangle list equally spaced over the region covered by the force field. The gaps between the triangles will allow the viewer to see the scene behind the triangles, as you might expect when looking at a force field. Apply a texture to the triangle list that gives the triangles the appearance of glowing with energy. For further information, see Triangle Lists and Current Texture.

Motion Blur

[This is preliminary documentation and subject to change.]

The perceived speed of an object in a 3-D scene can be enhanced by blurring the object, and by leaving a blurred trail of object images behind the object. Direct3D applications can accomplish this by rendering the object multiple times per frame.

Recall that Direct3D applications typically render scenes into an off-screen buffer. The contents of the buffer are displayed on the screen when the application calls the **IDirectDrawSurface4::Flip** method. Your Direct3D program can render the object multiple times into a scene before it displays the frame on the screen.

Programmatically, your application makes multiple calls to a DrawPrimitive method, repeatedly passing the same 3-D object. Before each call, the position of the object is updated slightly, producing a series of blurred object images on the target rendering

surface. If the object has one or more textures, your application can enhance the motion blur effect by rendering the first image of the object with all of its textures nearly transparent. Each time the object is rendered, the transparency of the object's texture is decreased. When your program renders the object in its final position, it should render the object's textures without transparency. The exception is if you're adding motion blur to another effect that requires texture transparency. In any case, the initial image of the object in the frame should be the most transparent. The final image should be the least transparent.

After your application renders the series of object images onto the target rendering surface and renders the rest of the scene, it should call the **IDirectDrawSurface4::Flip** method to display the frame on the screen.

If your program is simulating the effect of the user moving through a scene at high speed, it can add motion blur to the entire scene. In this case, your application would render the entire scene multiple times per frame. Each time the scene is rendered, your program must move the viewpoint slightly. If your scene is highly complex, your user may see a visible performance degradation as acceleration is increased because of the increasing number of scene renderings per frame.

Stencil Buffer Techniques

[This is preliminary documentation and subject to change.]

Applications use the stencil buffer to mask pixels in an image. The mask controls whether or not the pixel is drawn. For more information on the stencil buffer, see Stencil Buffers.

Direct3D applications can achieve a wide range of special effects with the stencil buffer. Some of the more common effects are listed below. This list is by no means exhaustive.

- Dissolves, Fades, and Swipes
- Decaling
- Compositing
- Outlines and Silhouettes

Dissolves, Fades, and Swipes

[This is preliminary documentation and subject to change.]

Applications are increasingly employing special effects that are commonly used in movies and video, such as dissolves, swipes, and fades.

In a dissolve, one image is gradually replaced by another in a smooth sequence of frames. Although Direct3D provides methods of using multiple texture blending to achieve the same effect, applications that use the stencil buffer for dissolves can utilize the texture blending capabilities for other effects while they do a dissolve.

When your application performs a dissolve, it must render two different images. It uses the stencil buffer to control which pixels from each image are drawn to the rendering target surface. You can define a series of stencil masks and copy them into the stencil buffer on successive frames. Alternately, you can define a base stencil mask for the first frame and alter it incrementally.

At the beginning of the dissolve, your application sets the stencil function and stencil mask such that most of the pixels from the starting image pass the test. Most of the pixels of the ending image should fail the stencil test. On successive frames, the stencil mask is updated so that fewer and fewer of the pixels in the starting image pass the test. As the frames progress, fewer and fewer of the pixels in the ending image fail the test. In this manner, your application can perform a dissolve using any arbitrary dissolve pattern.

Fading in or fading out is just a special case of dissolving. When fading in, the stencil buffer is used to dissolve from a black (or white) image to a rendering of a 3-D scene. Fading out is the opposite, your application starts with a rendering of a 3-D scene and dissolves to black (or white). The fade can be done using any arbitrary pattern you want to employ.

Direct3D applications use a similar technique for swipes. When an application performs, for instance, a left-to-right swipe. The ending image appears to gradually slide on top of the starting image from left to right. just as in doing a dissolve, you must define a series of stencil masks that are loaded into the stencil buffer on successive frames, or successively modify the starting stencil mask. The stencil masks are used to disable the writing of pixels from the starting image and enable the writing of pixels from the ending image.

A swipe is somewhat more complex than a dissolve in that your application must read pixels from the ending image in the reverse order of the swipe. That is, if the swipe is moving left to right, your application must read pixels from the ending image from right to left.

Decaling

[This is preliminary documentation and subject to change.]

Direct3D applications use the technique of decaling to control which pixels from a particular primitive image are drawn to the rendering target surface. Programs apply decals to the images of primitives to enable co-planar polygons to be rendered correctly.

For instance, when applying tire marks and yellow lines to a roadway, the markings should appear directly on top of the road. However, the z values of the markings and the road are the same. Therefore, the depth buffer might not produce a clean separation between the two. Some pixels in the back primitive may be rendered on top of the front primitive and visa versa. The resulting image appears to shimmer from frame to frame. This effect is called "Z Fighting" or "flimmering".

To solve this problem, use a stencil to mask out the section of the back primitive where the decal will appear. Turn off z-buffering and render the image of the front primitive into the masked-off area of the render target surface.

Although multiple texture blending can be used to solve this problem, doing so limits the number of other special effects that your application can produce. Using the stencil buffer to apply decals frees up texture blending stages for other effects.

Compositing

[This is preliminary documentation and subject to change.]

Your application can use the stencil buffer to composite 2-D or 3-D images onto a 3-D scene. A mask in the stencil buffer is used to occlude an area of the rendering target surface. Stored 2-D information, such as text or bitmaps, can then be written to the occluded area. Alternately, your application can render additional 3-D primitives to the stencil-masked region of the rendering target surface. It can even render an entire scene.

Games often composite multiple 3-D scenes together. For instance, driving games typically display a rear-view mirror. The mirror contains the view of the 3-D scene behind the driver. It is essentially a second 3-D scene composited together with the driver's forward view.

Outlines and Silhouettes

[This is preliminary documentation and subject to change.]

The stencil buffer can be used for more abstract effects, such as outlining and silhouetting.

If your program applies a stencil mask to the image of a primitive that is the same shape, but slightly smaller than the primitive, the resulting image will only contain the primitive's outline. It can then fill the stencil-masked area of the image with a solid color, giving the primitive an embossed look.

If the stencil mask is the same size and shape as the primitive you are rendering, the resulting image will contain a "hole" where the primitive should be. Your program can then fill the "hole" with black to produce a silhouette of the primitive.

Colored Lights

[This is preliminary documentation and subject to change.]

The **dcvColor** member of the **D3DLIGHT2** structure specifies a **D3DCOLORVALUE** structure. The colors defined by this structure are RGBA values that generally range from zero to one, with zero being black. Although you will usually want the light color to fall within this range, you can use values outside the range for special effects. For example, you could create a strong light that washes out a scene by setting the color to large values. You could also set the color to negative values to create a dark light, which actually removes light from a scene.

Dark lights are useful for forcing dramatic shadows in scenes and other special effects.

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

Antialiasing

[This is preliminary documentation and subject to change.]

Antialiasing is a technique you can use to reduce the appearance of "jaggies" — the stair-step pixels used to draw any line that isn't exactly horizontal or vertical. In three-dimensional scenes, this artifact is most noticeable on the boundaries between polygons of different colors. Antialiasing effectively blends the pixels at these boundaries to produce a more natural look to the scene.

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

- Edge antialiasing
- Full-scene antialiasing

Edge Antialiasing

[This is preliminary documentation and subject to change.]

In edge antialiasing, you render a scene, then re-render the convex silhouettes of the objects to be antialiased with lines. The system redraws those edges, blurring them to reduce artifacts.

If a device supports edge antialiasing, it exposes the D3DPRASTERCAPS_ANTIALIASEDGES capability flag in the **D3DPRIMCAPS** structure (filled by calling the **IDirect3DDevice3::GetCaps** method). If it does, set the D3DRENDERSTATE_EDGEANTIALIAS flag to TRUE, then redraw only the edges in the scene, using **IDirect3DDevice3::DrawPrimitive** and either the D3DPT_LINESTRIP or D3DPT_LINELIST primitive type. The behavior of edge antialiasing is undefined for primitives other than lines, so make sure to disable the feature by setting D3DRENDERSTATE_EDGEANTIALIAS to FALSE when antialiasing is complete.

Redrawing every edge in your scene can work without introducing major artifacts, but it can be computationally expensive. In addition, it can be difficult to determine which edges should be antialiased. The most important edges to redraw are those between areas of very different color (for example, silhouette edges) or boundaries between very different materials. Antialiasing the edge between two polygons of roughly the same color will have no effect, yet is still computationally expensive. For

these reasons, full-scene antialiasing is often preferred, given that the current hardware supports it. For more information, see Full-scene Antialiasing.

Full-scene Antialiasing

[This is preliminary documentation and subject to change.]

Full-scene antialiasing refers to blurring the edges of each polygon in the scene as it is rasterized in a single pass — no second pass is required. Note that full-scene antialiasing, when supported, only affects triangles and groups of triangles; lines cannot be antialiased by using Direct3D services.

On some hardware, full-scene antialiasing can be applied only when the application renders the polygons sorted from back to front. To find out whether this is true for the current device, retrieve the device capabilities by calling **IDirect3DDevice3::GetCaps** method. After the call, check the **dwRasterCaps** member of the associated **D3DPRIMCAPS** structures. If the device requires back to front rendering for antialiasing, it exposes the D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT capability flag in **dwRasterCaps**. If the device can perform antialiasing without regard to polygon order, it will expose the D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT flag. Of course, the absence of both flags indicates that the device cannot perform full-scene antialiasing at all.

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

When you no longer need full-scene antialiasing, disable it by setting D3DRENDERSTATE_ANTIALIAS to D3DANTIALIAS_NONE.

GUIDs

[This is preliminary documentation and subject to change.]

Direct3D uses globally unique identifiers, or GUIDs, to identify parts of the interface. When you use the **QueryInterface** method to determine whether an object supports an interface, you identify the interface you're interested in by using its GUID.

To use GUIDs successfully in your application, you must either define INITGUID prior to all other include and define statements, or you must link to the Dxguid.lib library. You should define INITGUID in only one of your source modules.

Note that you use GUIDs differently depending on whether your application is written in C or C++. In C, you pass a pointer to the GUID (&IID_IDirect3D, for example), but in C++, you pass a reference to it (simply IID_IDirect3D).

Performance Optimization

[This is preliminary documentation and subject to change.]

Every developer who creates real-time applications that use 3-D graphics is concerned about performance optimization. This section provides you with guidelines about getting the best performance from your code.

You can use the guidelines in the following sections for any Direct3D application:

- Databases and Culling
- Batching Primitives
- Lighting Tips
- Texture Size
- General Performance Tips
- Ramp Performance Notes
- Z-buffer Performance

Databases and Culling

[This is preliminary documentation and subject to change.]

Building a reliable database of the objects in your world is the key to excellent performance in Direct3D—it is more important than improvements to rasterization or hardware.

You should maintain the lowest polygon count you can possibly manage. Design for a low polygon count, building low-poly models from the start, and add polygons if you feel that you can do so without sacrificing performance later in the development process. Try to keep the total number of polygons in the neighborhood of 2500. Remember, "the fastest polygons are the ones you don't draw."

Batching Primitives

[This is preliminary documentation and subject to change.]

To get the best rendering performance during execution, you should try to work with primitives in batches and keep the number of render-state changes as low as possible. For example, if you have an object with two textures, group the triangles that use the first texture and follow them with the necessary render state to change the texture, then group all the triangles that use the second texture. The simplest hardware support for Direct3D is called with batches of render states and batches of primitives through the hardware abstraction layer (HAL). The more effectively the instructions are batched, the fewer HAL calls are performed during execution.

Lighting Tips

[This is preliminary documentation and subject to change.]

Since lights add a per-vertex cost to each rendered frame, you can achieve significant performance improvements by being careful about how you use them in your application. Most of the following tips derive from the maxim, "the fastest code is code that is never called."

- Use as few lights as possible. If you just need to bring up the overall level of lighting, use the ambient light instead of adding a new light. (It's much cheaper.)
- Directional lights are cheaper than point lights or spotlights. For directional lights, the direction to the light is fixed and doesn't need to be calculated on a per-vertex basis.
- Spotlights can be cheaper than point lights, because the area outside of the cone of light is calculated quickly. Whether or not they are cheaper depends on how much of your scene is lit by the spotlight.
- Use the range parameter to limit your lights to only the parts of the scene you need to illuminate. All the light types exit fairly early when they are out of range.
- Specular highlights almost double the cost of a light — use them only when you must. Use the D3DLIGHT_NO_SPECULAR flag in the **D3DLIGHT2** structure as often as reasonable. When defining materials you must set the specular power value to zero to turn off specular highlights for that material — simply setting the specular color to 0,0,0 is not enough.

Texture Size

[This is preliminary documentation and subject to change.]

Texture-mapping performance is heavily dependent on the speed of memory. There are a number of ways to maximize the cache performance of your application's textures.

- Keep the textures small; the smaller the textures are, the better chance they have of being maintained in the main CPU's secondary cache.
- Do not change the textures on a per-primitive basis. Try to keep polygons grouped in order of the textures they use.
- Use square textures whenever possible. Textures whose dimensions are 256×256 are the fastest. If your application uses four 128×128 textures, for example, try to ensure that they use the same palette and place them all into one 256×256 texture. This technique also reduces the amount of texture swapping. Of course, you should not use 256×256 textures unless your application requires that much texturing because, as already mentioned, textures should be kept as small as possible.

General Performance Tips

[This is preliminary documentation and subject to change.]

You can follow a few general guidelines to increase the performance of your application.

- Only clear when you must.
- Minimize state changes.
- Use perspective correction only if you must.
- If you can use smaller textures, do so.
- Gracefully degrade special effects that require a disproportionate share of system resources.
- Constantly test your application's performance.
- Ensure that your application runs well both with hardware acceleration and software emulation.

Ramp Performance Notes

[This is preliminary documentation and subject to change.]

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

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

Note

The ramp software emulation driver is obsolete, and not supported in DirectX 6.0 and later. You cannot create a Direct3D ramp device by using the **IDirect3D3** interface, nor can you query an existing ramp device for the **IDirect3DDevice3** interface. In short, ramp devices do not support any multiple texture blending options. The notes in this section apply only to using the ramp device with the **IDirect3D2**, **IDirect3DDevice2** or earlier interfaces.

Ramp Textures

[This is preliminary documentation and subject to change.]

Applications that use the ramp driver should be conservative with the number of texture colors they require. Each color used in a monochromatic texture requires its own lookup table during rendering. If your application uses hundreds of colors in a scene during rendering, the system must use hundreds of lookup tables, which do not cache well. Also, try to share palettes between textures whenever possible. Ideally, all

of your application's textures will fit into one palette, even when you are using a ramp driver with depths greater than 8-bit color.

Copy Texture-blending Mode

[This is preliminary documentation and subject to change.]

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

Copy mode is the simplest form of rasterization and hence the fastest. When copy mode rasterization is used, no lighting or shading is performed on the texture. The bytes from the texture are copied directly to the screen and mapped onto polygons using the texture coordinates in each vertex. Hence, when using copy mode, your application's textures must use the same pixel format as the primary surface. They must also use the same palette as the primary surface.

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

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

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

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

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

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

Ramp Performance Tips

[This is preliminary documentation and subject to change.]

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

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

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

Z-Buffer Performance

[This is preliminary documentation and subject to change.]

Applications can increase performance when using z-buffering and texturing by ensuring that scenes are rendered from front to back. Textured z-buffered primitives are pretested against the z-buffer on a scan line basis. If a scan line is hidden by a previously rendered polygon, the system rejects it quickly and efficiently. Z-buffering can improve performance, but the technique is most useful when a scene includes a great deal of *overdraw*. Overdraw is the average number of times a screen pixel is written to. Overdraw is difficult to calculate exactly, but you can often make a close approximation. If the overdraw averages less than 2, you can achieve the best performance by turning z-buffering off, and rendering the scene from back-to-front.

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

connected by an open door. If you render the first room and then draw the doorway to the second room using a z-test polygon, you may discover that the doorway is hidden by one of the objects in the first room and that you don't need to render anything at all in the second room.

On faster personal computers, software rendering to system memory is often faster than rendering to video memory, although it has the disadvantage of not being able to use double buffering or hardware-accelerated clear operations. If your application can render to either system or video memory, and if you include a routine that tests which is faster, you can take advantage of the best approach on the current system. The Direct3D sample code in this SDK demonstrates this strategy. It is necessary to implement both methods because there is no other way to test the speed. Speeds can vary enormously from computer to computer, depending on the main-memory architecture and the type of graphics adapter being used.

Troubleshooting

[This is preliminary documentation and subject to change.]

This section lists common categories of problems that you may encounter when writing Direct3D programs, and what you should do to prevent them.

- Device Creation
- Nothing Visible
- Debugging
- Borland Floating-Point Initialization
- Parameter Validation
- Miscellaneous

Device Creation

[This is preliminary documentation and subject to change.]

If your application fails during device creation, check for the following common errors:

- You must specify DDSCAPS_3DDEVICE when you create the DirectDraw surface.
- If you're using a palettized device, you must attach the palette.
- If you're using a z-buffer, you must attach it to the rendering target.
- Make sure you check the device capabilities, particularly the render depths.
- Check whether you are using system or video memory.
- Ensure that the registry has not been corrupted.

Nothing Visible

[This is preliminary documentation and subject to change.]

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

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

Debugging

[This is preliminary documentation and subject to change.]

Debugging a Direct3D application can be challenging. In addition to checking all the return values (a particularly important piece of advice in Direct3D programming, which is so dependent on very different hardware implementations), try the following techniques:

- Switch to debug DLLs.
- Force a software-only device, turning off hardware acceleration even when it is available.
- Force surfaces into system memory.
- Create an option to run in a window, so that you can use an integrated debugger.

The second and third options in the preceding list can help you avoid the Win16 lock which can otherwise cause your debugger to hang.

Also, try adding the following entries to WIN.INI:

```
[Direct3D]
debug=3
[DirectDraw]
debug=3
```

Borland Floating-Point Initialization

[This is preliminary documentation and subject to change.]

Compilers from the Borland company report floating-point exceptions in a manner that is incompatible with Direct3D. To solve this problem, you should include a _matherr() exception handler like the following:

```
// Borland floating point initialization
#include <math.h>
#include <float.h>

void initfp(void)
{
   // disable floating point exceptions
   _control87(MCW_EM,MCW_EM);
}

int _matherr(struct _exception  *e)
{
   e;              // dummy reference to catch the warning
   return 1;    // error has been handled
}
```

Parameter Validation

[This is preliminary documentation and subject to change.]

For performance reasons, the debug version of the Direct3D Immediate Mode runtime performs more parameter validation than the retail version, which sometimes performs no validation at all. This allows applications to perform robust debugging with the slower debug runtime component before using the faster retail version for performance tuning and final release.

Although several Direct3D Immediate Mode methods impose limits on the values that they can accept, these limits are often only checked and enforced by the debug version of the Direct3D Immediate Mode runtime. Applications must conform to these limits, or unpredictable (and highly undesirable) results can occur when running on the retail version of Direct3D. For example, the **IDirect3DDevice3::DrawPrimitive** method accepts a parameter (*dwVertexCount*) that indicates the number of vertices that the method will render. The method can only accept values between 0 and 65,535 (0x0000 and 0xFFFF). In the debug version of Direct3D, if you pass 65,536 (one more than the limit), the method will fail gracefully, printing an error message to the error log, and returning an error value to your application. Conversely, if your application makes the same error when it is running with the retail version of the runtime, behavior is undefined. For performance reasons, the method does not validate the parameters, resulting in unpredictable and completely situational behavior when they are not valid. In some cases the call might work, and in other cases it might cause a memory fault in Direct3D. If an invalid call consistently works with a particular hardware configuration and DirectX version, there is no guarantee that it will continue to function on other hardware or with future releases of DirectX.

If your application encounters unexplained failures when running with the retail Direct3D Immediate Mode runtime, test against the debug version and look closely for cases where your application passes invalid parameters.

Miscellaneous

[This is preliminary documentation and subject to change.]

The following tips can help you uncover common miscellaneous errors:

- Check the memory type (system or video) for your textures.
- Verify that the current hardware can do texturing.
- Make sure that you can restore any lost surfaces.
- Always specify D3DLIGHTSTATE_MATERIAL, even in RGB mode, because it is always necessary in monochromatic mode.

Direct3D Immediate Mode Tutorials

[This is preliminary documentation and subject to change.]

This section contains a series of tutorials, each providing step-by-step instructions for implementing the basics of Direct3D® Immediate Mode in a C/C++ or Visual Basic application. The tutorials are written parallel to a set of sample files that are provided with this SDK in the \Samples\Multimedia\D3DIM\Tutorials directory, following their code path and providing explanations along the way. Readers are encouraged to follow along in the sample code as they move through these tutorials.

- Direct3D Immediate Mode C/C++ Tutorials
- Direct3D Immediate Mode Visual Basic Tutorials

Direct3D Immediate Mode C/C++ Tutorials

[This is preliminary documentation and subject to change.]

The tutorials in this section show how to use Direct3D for common tasks by dividing those tasks into required steps. In some cases, steps are organized into substeps for clarity. The following tutorials are presented here:

- Tutorial 1: Rendering a Single Triangle
- Tutorial 2: Adding a Depth Buffer

Note

The sample files in these tutorials are written in C++. If you are using a C compiler, you must make the appropriate changes to the files for them to successfully compile. At the very least, you need to add the *vtable* and *this* pointers to the interface methods.
Some comments in the included sample code might differ from the source files in the SDK. Changes are made for brevity only, and are limited to comments to avoid changing the behavior of the code.

Tutorial 1: Rendering a Single Triangle

[This is preliminary documentation and subject to change.]

To use Direct3D, you first create an application window, then create and initialize DirectDraw® and Direct3D-related objects. You use the COM interfaces that these objects implement to manipulate them and to create the subordinate objects required to render a scene. The Triangle sample project upon which this tutorial is based illustrates these tasks by rendering the simplest possible scene: a single triangle. The Triangle sample uses the following steps to set up Direct3D, render a scene, and eventually shut down:

- Step 1: Create a Window
- Step 2: Initialize System Objects
- Step 3: Initialize the Scene
- Step 4: Monitor System Messages
- Step 5: Render and Display the Scene
- Step 6: Shut Down

In addition to these steps, Triangle performs some standard tasks common to windowed applications. Although these tasks aren't difficult, the Triangle sample (and all other windowed samples) performs the following tasks as the needed:

- Handle Window Movement
- Handle Window Resizing

Step 1: Create a Window

[This is preliminary documentation and subject to change.]

The first thing any Windows application must do when it is executed is create an application window to display a user interface. Keeping with this, when Triangle begins execution at its **WinMain** function, it uses the following code to perform window initialization:

```
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT )
{
    // Register the window class
```

```
WNDCLASS wndClass = { CS_HREDRAW | CS_VREDRAW, WndProc, 0, 0, hInst,
                      LoadIcon( hInst, MAKEINTRESOURCE(IDI_MAIN_ICON)),
                      LoadCursor(NULL, IDC_ARROW),
                      (HBRUSH)GetStockObject(WHITE_BRUSH), NULL,
                      TEXT("Render Window") };
RegisterClass( &wndClass );

// Create our main window
HWND hWnd = CreateWindow( TEXT("Render Window"),
                          TEXT("D3D Tutorial: Drawing One Triangle"),
                          WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                          CW_USEDEFAULT, 300, 300, 0L, 0L, hInst, 0L );
ShowWindow( hWnd, SW_SHOWNORMAL );
UpdateWindow( hWnd );
```

The preceding code is standard Windows programming, covered here mostly for thoroughness. The sample starts by defining and registering a window class called "Render Window." The window class is defined to redraw on size events, to use an application-provided icon as a resource, and to have a white background. After the class is registered, the code creates a basic top-level window that uses the registered class, with a client area of 300 pixels wide by 300 pixels tall, and has no menu or child windows. The sample uses the WS_OVERLAPPEDWINDOW window style to create a window that includes minimize, maximize, and close boxes common to windowed applications. (If the sample were to run in full-screen mode, the preferred window style is WS_EX_TOPMOST.) Once the window is created, the code calls standard Win32® functions to show and update the window.

With the application window ready, you can begin setting up the essential DirectX® objects, which is the topic of Step 2: Initialize System Objects.

Step 2: Initialize System Objects

[This is preliminary documentation and subject to change.]

After you create an application window, you can begin initializing the primary DirectX objects whose services you will draw on to render the scene. For a Direct3D application, this means creating and configuring DirectDraw, rendering surfaces, a rendering device, and a viewport. For clarity, the Triangle sample separates system object initialization code from the code for initializing the scene. As a result, geometry, application-specific data structures, and lesser Direct3D objects like materials are initialized with the scene. This isn't a requirement, but it makes for simpler code.

The Triangle sample performs system initialization in the Initialize3DEnvironment application-defined function, called from **WinMain** after the window is created. Although preparing these objects isn't complex, the code is a little too lengthy to discuss in one place. As a result, the steps taken by the Initialize3DEnvironment function are presented in the following substeps:

- Step 2.1: Initialize DirectDraw
- Step 2.2: Set Up DirectDraw Surfaces
- Step 2.3: Initialize Direct3D
- Step 2.4: Prepare the Viewport

Note

The Triangle sample code performs initialization by calling methods from within the **WinMain** function, immediately after the application window is created, rather than in response to system creation messages such as WM_CREATE. It does this to avoid relying on system message ordering, which can differ across platforms.

Step 2.1: Initialize DirectDraw

[This is preliminary documentation and subject to change.]

After creating the application window, the first object you will create is the DirectDraw object, which is required to set your application's cooperative level, and to create the surfaces for display and for use as the render target of a rendering device.

The Triangle sample starts performing initialization by creating a DirectDraw object and setting the application's cooperative level, as shown in the following code:

```
HRESULT Initialize3DEnvironment( HWND hWnd )
{
    HRESULT hr;

    // Create a DirectDraw object.
    hr = DirectDrawCreate( NULL, &g_pDD1, NULL );
    if( FAILED( hr ) )
        return hr;

    // Get a ptr to an IDirectDraw4 interface. This interface to DirectDraw
    // represents the DX6 version of the API.
    hr = g_pDD1->QueryInterface( IID_IDirectDraw4, (VOID**)&g_pDD4 );
    if( FAILED( hr ) )
        return hr;
```

The preceding code creates a DirectDraw object by calling the **DirectDrawCreate** DirectDraw global function. It passes NULL in the first parameter to request that the function create a DirectDraw object for the active display driver. For hardware that doesn't support GDI, such as 3-D only hardware, you should explicitly specify the globally unique identifier (GUID) of the desired driver in the first parameter. These GUIDs are normally obtained through enumeration. The second parameter is the address of a global variable that **DirectDrawCreate** fills with the address of the **IDirectDraw** interface for the DirectDraw object, and the last parameter is set to NULL to indicate that the new object will not be used with COM aggregation

features. If the DirectDraw object is created successfully, the code queries for the latest iteration of the DirectDraw interface, **IDirectDraw4**.

The sample continues by setting the application's cooperative level, as follows:

```
hr = g_pDD4->SetCooperativeLevel( hWnd, DDSCL_NORMAL );
if( FAILED( hr ) )
   return hr;
```

The sample sets the cooperative level by calling the **IDirectDraw4::SetCooperativeLevel** method. Setting the cooperative level effectively tells the system whether or not the application will render in full-screen mode or in a window. (Note that some hardware cannot render into a window. You can detect such hardware by checking for the absence of the DDCAPS2_CANRENDERWINDOWED capability flag when you call **IDirectDraw4::GetCaps**.) The code requests windowed cooperative level, also called the "normal" cooperative level, by including the DDSCL_NORMAL in the second parameter it passes to **SetCooperativeLevel**. The **SetCooperativeLevel** method can fail if another application already controls owns full-screen, exclusive mode.

Note

You can include the DDSCL_FPUSETUP cooperative level flag to increase performance. For more information about this cooperative level flag, see DirectDraw Cooperative Levels and FPU Precision. For general information, see Cooperative Levels in the DirectDraw documentation.

Once you create the DirectDraw object and set the cooperative level, you're ready to prepare the surfaces that will be used to contain and display a rendered scene. The Triangle sample does this as discussed in Step 2.2: Set Up DirectDraw Surfaces.

Step 2.2: Set Up DirectDraw Surfaces

[This is preliminary documentation and subject to change.]

After you create a DirectDraw object and set the cooperative level, you can create the surfaces that your application will use to render and display a scene. Exactly how you create your surfaces depends largely on whether or not your application will run in a window or in full-screen mode.

Full-screen Application Note

Applications that will run in full-screen mode can create surfaces as shown in the preceding code examples. More often, these applications should take advantage of page-flipping, a feature only available in full-screen, exclusive mode. In this case, instead of explicitly creating two surfaces, you can create a flipping chain of surfaces with a single call. For more information, see Creating Complex Surfaces and Flipping Chains.

The Triangle sample, designed to run in a window, starts by creating a primary surface, which represents the display:

```
// Prepare a surface description for the primary surface.
DDSURFACEDESC2 ddsd;
ZeroMemory( &ddsd, sizeof(DDSURFACEDESC2) );
ddsd.dwSize         = sizeof(DDSURFACEDESC2);
ddsd.dwFlags        = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

// Create the primary surface.
hr = g_pDD4->CreateSurface( &ddsd, &g_pddsPrimary, NULL );
if( FAILED( hr ) )
   return hr;
```

The description for the primary surface doesn't contain information about dimensions or pixel format, as these traits are assumed to be the same as the display mode. If the current display mode is 800x600, 16-bit color, DirectDraw ensures that the primary surface matches. After creating the primary surface, you can create the render target surface. In the case of Triangle, this is a separate off-screen surface created as follows:

```
ddsd.dwFlags        = DDSD_WIDTH | DDSD_HEIGHT | DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_3DDEVICE;

// Set the dimensions of the back buffer. Note that if our window changes
// size, we need to destroy this surface and create a new one.
GetClientRect( hWnd, &g_rcScreenRect );
GetClientRect( hWnd, &g_rcViewportRect );
ClientToScreen( hWnd, (POINT*)&g_rcScreenRect.left );
ClientToScreen( hWnd, (POINT*)&g_rcScreenRect.right );
ddsd.dwWidth  = g_rcScreenRect.right - g_rcScreenRect.left;
ddsd.dwHeight = g_rcScreenRect.bottom - g_rcScreenRect.top;

// Create the back buffer. The most likely reason for failure is running
// out of video memory. (A more sophisticated app should handle this.)
hr = g_pDD4->CreateSurface( &ddsd, &g_pddsBackBuffer, NULL );
if( FAILED( hr ) )
   return hr;
```

The preceding code creates an off-screen surface that is equal to the dimensions of the program window. There is no need to create a larger surface, because the dimensions of the window dictate what is visible to the user. (This code also initializes two global variables that are later used to set up the viewport and track the application window size and position.) As the preceding excerpt shows, you must include the DDSCAPS_3DDEVICE capability for any surface that will be used as a render target. This capability causes the system to allocate additional internal data structures that are used only for 3-D rendering. As when creating the primary surface, the pixel format for the off-screen surface is assumed to be the same as the display mode when it isn't provided in the surface description.

Note

Applications that will use a depth buffer should create one and attach it to the render target surface at this point. For simplicity, this tutorial doesn't employ a depth buffer, but they are covered in Tutorial 2: Adding a Depth Buffer and in Depth Buffers.

After creating the primary and render target surface, you can create and attach a DirectDrawClipper object to the display surface. Using a clipper frees you from attempting to handle cases when the window is partially obscured by other windows, or when the window is partially outside the display area. Clippers aren't needed for applications that run in full-screen mode. The Triangle sample uses the following code to create a clipper and associate it with the display window:

```
LPDIRECTDRAWCLIPPER pcClipper;

// Create the clipper.
hr = g_pDD4->CreateClipper( 0, &pcClipper, NULL );
if( FAILED( hr ) )
   return hr;

// Assign it to the window handle, then set
// the clipper to the desired surface.
pcClipper->SetHWnd( 0, hWnd );
g_pddsPrimary->SetClipper( pcClipper );
pcClipper->Release();
```

Having created the basic DirectDraw objects, you can move on to setting up the essential Direct3D objects that will render the scene. The Triangle sample performs this task in Step 2.3: Initialize Direct3D.

Step 2.3: Initialize Direct3D

[This is preliminary documentation and subject to change.]

After you create the surfaces your application will need to render and display a scene, you can begin initializing Direct3D objects by retrieving a pointer to the **IDirect3D3** interface for the DirectDraw object, which is used to create all the objects you'll need to render a scene. Note that this interface is exposed by the DirectDraw object, and represents a separate set of features, not a separate object. You retrieve the **IDirect3D3** interface by calling the **IUnknown::QueryInterface** method of the DirectDraw object. The following code from Triangle performs this task:

```
// Query DirectDraw for access to Direct3D
g_pDD4->QueryInterface( IID_IDirect3D3, (VOID**)&g_pD3D );
if( FAILED( hr) )
   return hr;
```

After retrieving a pointer to the IDirect3D3 interface, you can create a rendering device by calling the **IDirect3D3::CreateDevice** method. The **CreateDevice** method accepts the globally unique identifier (GUID) of the desired device, the address of the **IDirectDrawSurface4** interface for the surface that the device will render to, and the address of a variable that the method will set to an **IDirect3DDevice3** interface pointer if the device object is created successfully. Although the tutorial uses hard-coded GUID values, a real application should enumerate devices to get a GUID. For information about device enumeration, see Enumerating Direct3D Devices.

(The Triangle sample checks the display mode prior to creating the device. If the display is set to a palettized mode, it exits. Attempting to create a device for a palettized surface that doesn't have an associated palette will cause the **CreateDevice** method to fail. This is done for simplicity. A real-world application should create a render target surface and attach a palette, or require that the user set their display mode to 16-bit color or higher.)

The following code, taken from Triangle, checks the display mode, and creates a rendering device:

```
// Check the display mode, and
ddsd.dwSize = sizeof(DDSURFACEDESC2);
g_pDD4->GetDisplayMode( &ddsd );
if( ddsd.ddpfPixelFormat.dwRGBBitCount <= 8 )
   return DDERR_INVALIDMODE;

// The GUID here is hardcoded. In a real-world application
// this should be retrieved by enumerating devices.
hr = g_pD3D->CreateDevice( IID_IDirect3DHALDevice,
                           g_pddsBackBuffer,
                           &g_pd3dDevice, NULL );
if( FAILED( hr ) )
{
   // If the hardware GUID doesn't work, try a software device.
   hr = g_pD3D->CreateDevice(IID_IDirect3DRGBDevice,
                             g_pddsBackBuffer,
                             &g_pd3dDevice, NULL );
    if( FAILED( hr ) )
       return hr;
}
```

The **IDirect3D3::CreateDevice** method can fail for many reasons. The most likely cause is when the primary display device doesn't support 3-D features. Another possibility is if the display hardware cannot render in the current display mode. These possibilities should be checked during device enumeration. To keep the code simple, Triangle attempts to create a software rendering device if the hardware device cannot be created.

Note

Even though the **CreateDevice** method accepts a pointer to a DirectDrawSurface object, a rendering device is not a surface. Rather, it is a discrete COM object that uses a surface to contain graphics for a rendered scene.

After the device is created, you can create a viewport object and assign it to the device, as described in Step 2.4: Prepare the Viewport.

Step 2.4: Prepare the Viewport

[This is preliminary documentation and subject to change.]

After you create a rendering device, you can create a viewport object and assign it to the device. In short, the viewport determines how the geometry in a 3-D scene is clipped and then represented in the 2-D space of a display screen. (For a conceptual overview about viewports, see Viewports and Clipping.)

Setting up a viewport is a straight-forward process that starts with preparing the viewport parameters in a **D3DVIEWPORT2** structure. The Triangle sample sets the viewport parameters to the dimensions of the render target surface, with a standard 3-D clipping region that exists from -1.0 to 1.0 in x, from 1.0 to -1.0 in y, and from 0.0 to 1.0 in z:

```
// Set up the viewport data parameters
D3DVIEWPORT2 vdData;
ZeroMemory( &vdData, sizeof(D3DVIEWPORT2) );

// Always set the structure size!
vdData.dwSize       = sizeof(D3DVIEWPORT2);
vdData.dwWidth      = g_rcScreenRect.right - g_rcScreenRect.left;
vdData.dwHeight     = g_rcScreenRect.bottom - g_rcScreenRect.top;
vdData.dvClipX      = -1.0f;
vdData.dvClipWidth  = 2.0f;
vdData.dvClipY      = 1.0f;
vdData.dvClipHeight = 2.0f;
vdData.dvMaxZ       = 1.0f;
```

Once the viewport parameter structure is ready, Triangle creates the viewport and assigns it to the rendering device. Note that it doesn't actually apply the parameters until after the viewport is assigned to the device. This is a requirement.

```
// Create the viewport.
hr = g_pD3D->CreateViewport( &g_pvViewport, NULL );
if( FAILED( hr ) )
    return hr;

// Associate the viewport with the device.
g_pd3dDevice->AddViewport( g_pvViewport );

// Set the parameters for the new viewport.
```

```
g_pvViewport->SetViewport2( &vdData );
```

Assigning the viewport to the device merely adds it to an internal list of viewports for the device, it doesn't actually select the viewport to be used during rendering. The following code selects the viewport:

```
// Set the current viewport for the device
g_pd3dDevice->SetCurrentViewport( g_pvViewport );
```

Now that the basic DirectX objects have been created, you can start preparing the subordinate objects required to render scene, which is the topic of Step 3: Initialize the Scene.

Step 3: Initialize the Scene

[This is preliminary documentation and subject to change.]

After creating the primary Direct3D-related objects—a DirectDraw object, a rendering device, and a viewport—you can begin initializing the scene by setting up geometry, preparing materials, and configuring the transformation matrices your application will use. The Triangle sample uses an application-defined function, App_InitDeviceObjects, to perform the following steps that initialize the scene:

- Step 3.1: Prepare Geometry
- Step 3.2: Set Up Material and Initial Lighting States
- Step 3.3: Prepare and Set Transformation Matrices

Step 3.1: Prepare Geometry

[This is preliminary documentation and subject to change.]

After you create the primary system objects used with DirectDraw and Direct3D, you can begin initializing the scene. The Triangle sample takes this opportunity to initialize geometry by defining vertices in an array of **D3DVERTEX** structure. Technically, you aren't required to set up the geometry at this time—you can do it anytime prior to calling rendering methods:

```
HRESULT App_InitDeviceObjects( LPDIRECT3DDEVICE3 pd3dDevice,
                               LPDIRECT3DVIEWPORT3 pvViewport )
{
    // Data for the geometry of the triangle. Note that this tutorial only
    // uses ambient lighting, so the vertices' normals are not actually used.
    D3DVECTOR p1( 0.0f, 3.0f, 0.0f );
    D3DVECTOR p2( 3.0f,-3.0f, 0.0f );
    D3DVECTOR p3(-3.0f,-3.0f, 0.0f );
    D3DVECTOR vNormal( 0.0f, 0.0f, 1.0f );

    // Initialize the 3 vertices for the front of the triangle
    g_pvTriangleVertices[0] = D3DVERTEX( p1, vNormal, 0, 0 );
```

```
g_pvTriangleVertices[1] = D3DVERTEX( p2, vNormal, 0, 0 );
g_pvTriangleVertices[2] = D3DVERTEX( p3, vNormal, 0, 0 );

// Initialize the 3 vertices for the back of the triangle
g_pvTriangleVertices[3] = D3DVERTEX( p1, -vNormal, 0, 0 );
g_pvTriangleVertices[4] = D3DVERTEX( p3, -vNormal, 0, 0 );
g_pvTriangleVertices[5] = D3DVERTEX( p2, -vNormal, 0, 0 );
```

The preceding code fragment defines three points in 3-D space that define a triangle that sits upright in the z=0 plane. After defining the geometry to be displayed, the Triangle sample prepares material and lighting parameters in Step 3.2: Set Up Material and Initial Lighting States.

Step 3.2: Set Up Material and Initial Lighting States

[This is preliminary documentation and subject to change.]

After you create the basic 3-D rendering objects (a DirectDraw object, a rendering device, and a viewport), you've got almost all you need to render a simple scene. The next thing to do is to create and configure a material and set some initial lighting states. These can all be changed later if needed. For an introduction to these concepts, see Lighting and Materials.

The Triangle sample starts by retrieving the **IDirect3D3** interface pointer for the rendering device by calling the **IDirect3DDevice3::GetDirect3D** method (a pointer to the rendering device, and one to the viewport are passed to the App_InitDeviceObjects function from which this code is taken). Then, it creates a material object and sets material parameters, as shown here:

```
// Create the material object.
if( FAILED( pD3D->CreateMaterial( &g_pmtrlObjectMtrl, NULL ) ) )
    return E_FAIL;

// Set properties for ambient reflectance.
D3DMATERIAL       mtrl;
D3DMATERIALHANDLE hmtrl;
ZeroMemory( &mtrl, sizeof(D3DMATERIAL) );

// Always set the structure size!
mtrl.dwSize       = sizeof(D3DMATERIAL);
mtrl.dcvAmbient.r = 1.0f;
mtrl.dcvAmbient.g = 1.0f;
mtrl.dcvAmbient.b = 0.0f;

// Commit the properties to the material.
g_pmtrlObjectMtrl->SetMaterial( &mtrl );
```

The preceding code creates a material object, represented by the **IDirect3DMaterial3** interface, by calling the **IDirect3D3::CreateMaterial** object. The new interface is assigned to an application-defined global variable, *g_pmtrlObjectMtrl*. When created, a new material has no properties and cannot be used in rendering. The code prepares material properties in a **D3DMATERIAL** structure to describe a material that will reflect the red and green components of ambient light, making it appear yellow in the scene. (This tutorial only uses ambient light, so it only sets an ambient reflectance property. A real-world application would use direct light as well as ambient light, and should set therefore set diffuse and specular reflectance properties as well.) After preparing the material properties, the code applies them to the material object by calling the **IDirect3DMaterial3::SetMaterial** method.

Note

When using textures, the object material is usually omitted or colored white.

After setting the material properties, you can retrieve the material handle that is used to select it for rendering. You retrieve a material handle by calling the **IDirect3DMaterial3::GetHandle** method. Once you have the material handle, you can select the current material by calling the **IDirect3DDevice3::SetLightState** method, passing the D3DLIGHTSTATE_MATERIAL enumerated value in the first parameter and the material handle in the second parameter. The Triangle sample does this with the following code:

```
// Bind the material to the device.
g_pmtrlObjectMtrl->GetHandle( pd3dDevice, &hmtrl );

// Select the current material.
pd3dDevice->SetLightState(  D3DLIGHTSTATE_MATERIAL, hmtrl );
```

After the current material is selected, all polygons will be rendered using this material. However, before anything in the scene will be visible you need to provide some light. The code in Triangle sets an ambient light level by calling the **IDirect3DDevice3::SetLightState** method to set the D3DLIGHTSTATE_AMBIENT light state for white ambient light:

```
// Set up white ambient light.
pd3dDevice->SetLightState(  D3DLIGHTSTATE_AMBIENT,  0xffffffff );
```

Now that the geometry, material, and initial lighting parameters are set, the sample moves on to setting up the transformation matrices. This is covered in Step 3.3: Prepare and Set Transformation Matrices.

Step 3.3: Prepare and Set Transformation Matrices

[This is preliminary documentation and subject to change.]

Another step in setting up a simple scene involved setting the world, view, and projection matrices. The system applies these matrices to geometry to place it in the scene, adjust for the camera's location and orientation, and scale vertex data to make

distant objects appear smaller than near objects. (For a conceptual overview, see The Geometry Pipeline.)

The Triangle sample starts by creating an identity matrix in a **D3DMATRIX** structure, then it manipulates the matrix to produce the desired transformations. Once a matrix is ready, the code assigns it to the device by calling the **IDirect3DDevice3::SetTransform** method with the D3DTRANSFORMSTATE_WORLD, D3DTRANSFORMSTATE_VIEW, or D3DTRANSFORMSTATE_PROJECTION values:

```
// Start by setting up an identity matrix.
D3DMATRIX mat;
mat._11 = mat._22 = mat._33 = mat._44 = 1.0f;
mat._12 = mat._13 = mat._14 = mat._41 = 0.0f;
mat._21 = mat._23 = mat._24 = mat._42 = 0.0f;
mat._31 = mat._32 = mat._34 = mat._43 = 0.0f;

// The world matrix controls the position and orientation
// of the polygons in world space. We'll use it later to
// spin the triangle.
D3DMATRIX matWorld = mat;
pd3dDevice->SetTransform( D3DTRANSFORMSTATE_WORLD, &matWorld );

// The view matrix defines the position and orientation of
// the camera. Here, we are just moving it back along the z-
// axis by 10 units.
D3DMATRIX matView = mat;
matView._43 = 10.0f;
pd3dDevice->SetTransform( D3DTRANSFORMSTATE_VIEW, &matView );

// The projection matrix defines how the 3-D scene is "projected"
// onto the 2-D render target surface. For more information,
// see "What Is the Projection Transformation?"

// Set up a very simple projection that scales x and y
// by 2, and translates z by -1.0.
D3DMATRIX matProj = mat;
matProj._11 =  2.0f;
matProj._22 =  2.0f;
matProj._34 =  1.0f;
matProj._43 = -1.0f;
matProj._44 =  0.0f;
pd3dDevice->SetTransform( D3DTRANSFORMSTATE_PROJECTION, &matProj );
```

After setting the transformations, Triangle is done setting up the scene. The App_InitDeviceObjects application-defined function returns S_OK to the caller, the Initialize3DEnvironment application-defined function. The Initialize3DEnvironment

function then returns that value to **WinMain**, which moves on to process system messages, as discussed in Step 4: Monitor System Messages.

Step 4: Monitor System Messages

[This is preliminary documentation and subject to change.]

After you've created the application window, created the DirectX objects, then initialized the scene, you're ready to render the scene. In most cases Windows applications monitor system messages in their message loop, and render frames whenever no messages are in queue. The Triangle sample is no different; it uses the following code for its message loop:

```
BOOL bGotMsg;
MSG  msg;
PeekMessage( &msg, NULL, 0U, 0U, PM_NOREMOVE );
g_bReady = TRUE;

while( WM_QUIT != msg.message  )
{
   // Use PeekMessage() if the app is active, so we can use idle time to
   // render the scene. Else, use GetMessage() to avoid eating CPU time.
   if( g_bActive )
      bGotMsg = PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE );
   else
      bGotMsg = GetMessage( &msg, NULL, 0U, 0U );

   if( bGotMsg )
   {
         TranslateMessage( &msg );
         DispatchMessage( &msg );
   }
   else
   {
      // Render a frame during idle time (no messages are waiting)
      if( g_bActive && g_bReady )
         Render3DEnvironment();
   }
}
```

This code uses a global flag variable, *g_bActive*, to keep track of when it's active, and another variable, *g_bReady*, to indicate that all system objects are ready to render a scene. The application sets g_*bActive* to FALSE whenever the window isn't visible, and it sets the *g_bReady* variable to FALSE whenever it needs to recreate the objects used to render the scene. (The latter situation is covered in Handle Window Resizing.)

If the application is active, it checks the message queue to see if there are any pending messages. If there are messages in queue, the code dispatches them like any other

Windows application. Otherwise, it calls the Render3DEnvironment application-defined function to render a frame of the scene, which is the topic of Step 5: Render and Display the Scene.

Step 5: Render and Display the Scene

[This is preliminary documentation and subject to change.]

Whenever your application isn't processing system messages, it can render a frame of a scene. The Triangle sample renders the scene in the Render3DEnvironment application-defined function called from **WinMain** whenever the message queue is empty. The Render3DEnvironment function subdivides the task into three substeps:

- Step 5.1: Update the Scene
- Step 5.2: Render the Scene
- Step 5.3: Update the Display

Step 5.1: Update the Scene

[This is preliminary documentation and subject to change.]

Immediately after it is called, the Render3DEnvironment application-defined function in the Triangle sample calls App_FrameMove (another application-defined function). The App_FrameMove function simply updates the world matrix that Direct3D applies to the geometry to reflect a rotation around the y-axis based on an internal count value, passed to the function in the *fTimeKey* parameter. Because the rotation is applied once per frame, the end result looks like the model is rotating in place.

```
HRESULT App_FrameMove( LPDIRECT3DDEVICE3 pd3dDevice, FLOAT fTimeKey )
{
    // For this simple tutorial, we are rotating the triangle about the y-axis.
    // To do this, just set up a 4x4 matrix defining the rotation, and set it
    // as the new world transform.
    D3DMATRIX matSpin;
    matSpin._11 = matSpin._22 = matSpin._33 = matSpin._44 = 1.0f;
    matSpin._12 = matSpin._13 = matSpin._14 = matSpin._41 = 0.0f;
    matSpin._21 = matSpin._23 = matSpin._24 = matSpin._42 = 0.0f;
    matSpin._32 = matSpin._32 = matSpin._34 = matSpin._43 = 0.0f;

    matSpin._11 = (FLOAT)cos( fTimeKey );
    matSpin._33 = (FLOAT)cos( fTimeKey );
    matSpin._13 = (FLOAT)sin( fTimeKey );
    matSpin._31 = (FLOAT)sin( fTimeKey );

    pd3dDevice->SetTransform( D3DTRANSFORMSTATE_WORLD, &matSpin );

    return S_OK;
}
```

In the real world, of course, your applications will do much more than apply a single rotation on a single model. (For more information rotation matrices, see Rotation in the 3-D Transformations section.)

After you update the geometry in the scene, you can render it to the render target surface, as the Triangle sample does in Step 5.2: Render the Scene.

Step 5.2: Render the Scene

[This is preliminary documentation and subject to change.]

Once the geometry has been updated to reflect the desired animation, you can render the scene. The Triangle sample takes a typical approach: the App_Render application-defined function called from the sample's Render3DEnvironment function starts by clearing the viewport:

```
HRESULT App_Render( LPDIRECT3DDEVICE3 pd3dDevice,
                    LPDIRECT3DVIEWPORT3 pvViewport,
                    D3DRECT* prcViewportRect )
{
    // Clear the viewport to a blue color.
    pvViewport->Clear2( 1UL, prcViewportRect, D3DCLEAR_TARGET, 0x000000ff,
                        0L, 0L );
```

The preceding code calls the **IDirect3DViewport3::Clear2** method to clear the viewport. The first two methods that the **Clear2** method accepts are the address of an array of rectangles that describe the areas on the render target surface to be cleared, and a value that informs the method how many rectangles from the array should be cleared. In most cases, you'll use a single rectangle that covers the entire render target. The third parameter determines the method's behavior. It can clear a render-target surface, an associated depth buffer, the stencil buffer, or any combination of the three. Because this tutorial doesn't use a depth buffer, D3DCLEAR_TARGET is the only flag used. The last three parameters are set to reflect clearing values for the render target, depth buffer, and stencil buffer. The Triangle sample sets the clear color for the render target surface to blue. Because the remaining parameters aren't used in this tutorial, the code just sets them to zero. The **Clear2** method will ignores them when the corresponding flag isn't present.

After clearing the viewport, the Triangle sample informs Direct3D that rendering will begin, renders the scene, then signals that rendering is complete, as shown here:

```
    // Begin the scene.
    if( FAILED( pd3dDevice->BeginScene() ) )
        return E_FAIL;

    // Draw the triangle using a DrawPrimitive() call.
    pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, D3DFVF_VERTEX,
                               g_pvTriangleVertices, 6, NULL );
```

```
    // End the scene.
    pd3dDevice->EndScene();

    return S_OK;
}
```

The **IDirect3DDevice3::BeginScene** and **IDirect3DDevice3::EndScene** methods signal to the system when rendering is beginning or is complete. You can only call rendering methods between calls to these methods. Even if rendering methods fail, you should call **EndScene** before calling **BeginScene** again.

After rendering the scene to the off-screen render target, you can update the user's display. The tutorial sample performs this in Step 5.3: Update the Display.

Step 5.3: Update the Display

[This is preliminary documentation and subject to change.]

Once a scene has been rendered to the render-target surface, you can show the results on screen. A windowed application usually does this by blitting the content of the render-target surface to the primary surface, and a full-screen application that employs page-flipping would simply flip the surfaces in the flipping chain. The Triangle sample uses the former method because it runs in a window, using the following code:

```
HRESULT ShowFrame()
{
    // The g_pddsPrimary variable will be NULL when
    // the application is in the middle of recreating
    // DirectDraw objects.
    if( NULL == g_pddsPrimary )
        return E_FAIL;

    // We are in windowed mode, so perform a blit from the backbuffer to the
    // correct position on the primary surface
    return g_pddsPrimary->Blt( &g_rcScreenRect, g_pddsBackBuffer,
                               &g_rcViewportRect, DDBLT_WAIT, NULL );
}
```

Note that the preceding application-defined function simply blits the entire contents of the render target surface to the window on the desktop. The tutorial tracks the destination rectangle for the blit in the *g_rcScreenRect* global variable. This rectangle is updated whenever the user moves the window, as covered in the Handle Window Movement section.

Step 6: Shut Down

[This is preliminary documentation and subject to change.]

At some point during execution, your application must shut down. Shutting down a DirectX application not only means that you should destroy the application window, but you also deallocate any DirectX objects your application uses and invalidate the pointers to them. The Triangle calls an application-defined function to handle this cleanup, called Cleanup3DEnvironment, when it receives a WM_DESTROY message:

```
HRESULT Cleanup3DEnvironment()
{
    // Cleanup any objects created for the scene
    App_DeleteDeviceObjects( g_pd3dDevice, g_pvViewport );

    // Release the DDraw and D3D objects used by the app
    if( g_pvViewport )     g_pvViewport->Release();
    if( g_pD3D )           g_pD3D->Release();
    if( g_pddsBackBuffer ) g_pddsBackBuffer->Release();
    if( g_pddsPrimary )    g_pddsPrimary->Release();
    if( g_pDD4 )           g_pDD4->Release();

    // Do a safe check for releasing the D3DDEVICE. RefCount should be zero.
    if( g_pd3dDevice )
        if( 0 < g_pd3dDevice->Release() )
            return E_FAIL;

    // Do a safe check for releasing DDRAW. RefCount should be zero.
    if( g_pDD1 )
        if( 0 < g_pDD1->Release() )
            return E_FAIL;

    g_pvViewport     = NULL;
    g_pd3dDevice     = NULL;
    g_pD3D           = NULL;
    g_pddsBackBuffer = NULL;
    g_pddsPrimary    = NULL;
    g_pDD4           = NULL;
    g_pDD1           = NULL;

    return S_OK;
}
```

The preceding function deallocates the DirectX objects it uses by calling the **IUnknown::Release** methods for each object. Because the tutorial follows COM rules, the reference counts for most objects should become zero and are automatically removed from memory.

In addition to shut down, there are times during normal execution—such as when the user changes the desktop resolution or color depth—when you might need to destroy

and re-create the DirectX objects in use. As a result, it's handy to keep your application's cleanup code in one place, which can be called when the need arises.

Handle Window Movement

[This is preliminary documentation and subject to change.]

Any DirectX application that runs in a window must track the position of the client area for the window so that blits to the window will appear on the desktop in the right place. Failing to track this results in graphics appearing outside the application window—a confusing sight for the user. The Triangle sample responds to the WM_MOVE messages that the system sends to the window procedure as shown here:

```
.
.
.
    case WM_MOVE:
        // Move messages need to be tracked to update the screen rects
        // used for blitting the backbuffer to the primary.
        if( g_bActive && g_bReady )
            OnMove( (SHORT)LOWORD(lParam), (SHORT)HIWORD(lParam) );
        break;
.
.
.
```

The *g_bActive* variable is set elsewhere in the window procedure to reflect whether or not the window is active—when the window is minimized it's set to FALSE. The *g_bReady* variable is TRUE except when the application is in the midst of re-creating the DirectX objects it uses. If these variables are both TRUE, the OnMove application-defined function gets called:

```
VOID OnMove( INT x, INT y )
{
    DWORD dwWidth  = g_rcScreenRect.right - g_rcScreenRect.left;
    DWORD dwHeight = g_rcScreenRect.bottom - g_rcScreenRect.top;
    SetRect( &g_rcScreenRect, x, y, x + dwWidth, y + dwHeight );
}
```

This function simply recalculates the size of the global variable, *g_rcScreenRect*, that the tutorial uses as the destination rectangle when it updates the display in Step 5.3: Update the Display.

Handle Window Resizing

[This is preliminary documentation and subject to change.]

Any DirectX application that runs in a window must respond to any WM_SIZE messages that the system sends. The render target surface is usually kept as small as possible to conserve memory, and the smallest size is usually the size of the window

client area. When window size increases, you must destroy the render target surface and the associated objects and re-create it at an appropriate size. Technically, an application could do this only when the window gets larger, and respond to situations when window size decreases by adjusting the viewport and decreasing the size of the blit accordingly. For simplicity, this tutorial destroys and re-creates the objects it uses whenever it receives a WM_SIZE message:

```
.
.
.
case WM_SIZE:
    // Check to see if we are losing or gaining our window.
    // Set the active flag to match.
    if( SIZE_MAXHIDE==wParam || SIZE_MINIMIZED==wParam )
        g_bActive = FALSE;
    else g_bActive = TRUE;

    // A new window size will require a new back buffer size. The
    // easiest way to achieve this is to release and re-create
    // everything. Note: if the window gets too big, we may run out
    // of video memory and need to exit. This simple app exits
    // without displaying error messages, but a real app would
    // behave itself much better.
    if( g_bActive && g_bReady )
    {
        g_bReady = FALSE;
        if( FAILED( Cleanup3DEnvironment() ) )
            DestroyWindow( hWnd );
        if( FAILED( Initialize3DEnvironment( hWnd ) ) )
            DestroyWindow( hWnd );
        g_bReady = TRUE;
    }
    break;
.
.
.
```

Note that the preceding code sets a global variable to communicate to other portions of the code that the DirectX objects in use are being invalidated. In addition, this code calls the application-defined Cleanup3DEnvironment function to destroy the objects, which is also called during application shut-down. Ending the application is discussed in Step 6: Shut Down.

Tutorial 2: Adding a Depth Buffer

[This is preliminary documentation and subject to change.]

Direct3D applications often rely on depth buffers to properly display objects in a scene. For a conceptual overview, see Depth Buffers. To use depth buffering, you must enumerate supported depth-buffer formats, create a depth-buffer surface, attach it to a render-target surface, and enable depth buffering for the rendering device.

This tutorial parallels the code in the Triangle sample project which uses the most commonly supported type of depth buffer, a z-buffer. The ZBuffer sample performs the following steps to use a z-buffer:

- Step 1: Enumerate Depth-Buffer Formats
- Step 2: Create the Depth Buffer
- Step 3: Attach the Depth Buffer
- Step 4: Enable Depth Buffering

Note

The code in the ZBuffer sample is nearly identical to the code in Triangle. This tutorial focuses only on the depth-buffer code unique to ZBuffer, and does not cover setting up Direct3D, rendering, shutting down, or handling Windows messages. For information on these tasks, see Tutorial 1: Rendering a Single Triangle.
Because some rendering devices require depth buffers to be located in particular places in memory, the system requires that you create and attach the depth-buffer surface to the render-target surface before you create a rendering device.

Step 1: Enumerate Depth-Buffer Formats

[This is preliminary documentation and subject to change.]

Before you can create a depth buffer you must determine what depth-buffer formats, if any, are supported by the rendering device. Call the **IDirect3D3::EnumZBufferFormats** method to enumerate the depth-buffer formats that the device supports. The ZBuffer sample uses the following code to enumerate depth-buffer formats:

```
//-----------------------------------------------------------------------
// Create the z-buffer AFTER creating the back buffer and BEFORE creating
// the d3ddevice.
//
// Note: before creating the z-buffer, apps may want to check the device
// caps for the D3DPRASTERCAPS_ZBUFFERLESSHSR flag. This flag is true for
// certain hardware that can do HSR (hidden-surface-removal) without a
// z-buffer. For those devices, there is no need to create a z-buffer.
//-----------------------------------------------------------------------

DDPIXELFORMAT ddpfZBuffer;  // Passing this as a VOID*

g_pD3D->EnumZBufferFormats( *pDeviceGUID,
```

```
                    EnumZBufferCallback, (VOID*)&ddpfZBuffer );
```

The **EnumZBufferFormats** method accepts the globally unique identifier (GUID) of the device for which the formats will be enumerated, the address of a callback function, and the address of an arbitrary data structure that will be passed to the callback function. The callback function you provide must conform to the **D3DEnumPixelFormatsCallback** function prototype. The system calls the specified callback function once for each supported depth-buffer format, unless the callback function returns D3DENUMRET_CANCEL. The ZBuffer sample processes callbacks as follows:

```
static HRESULT WINAPI EnumZBufferCallback( DDPIXELFORMAT* pddpf,
                                           VOID* pddpfDesired )
{
    // If this is ANY type of depth-buffer, stop.
    if( pddpf->dwFlags == DDPF_ZBUFFER )
    {
        memcpy( pddpfDesired, pddpf, sizeof(DDPIXELFORMAT) );

        // Return with D3DENUMRET_CANCEL to end the search.
        return D3DENUMRET_CANCEL;
    }

    // Return with D3DENUMRET_OK to continue the search.
    return D3DENUMRET_OK;
}
```

When the system calls the callback function, it passes the address of a **DDPIXELFORMAT** structure that describes the pixel format of the depth buffer. The **dwFlags** member will contain DDPF_ZBUFFER for any pixel formats that include depth-buffer bits. If so, the **dwZBufferBitDepth** member includes an integer that represents the number of bits in the pixel format reserved for depth information, and the **dwZBitMask** member masks the relevant bits.

For simplicity, this tutorial only uses z-buffers, which are the most common type of depth buffer. It ignores any other formats (such as DDPF_STENCILBUFFER) that the system enumerates. Applications could also check the bit depth of the z-buffer (8-, 16-, 24-, 32-bit) and make a choice based on that as well. If a suitable format is found, the function copies the provided **DDPIXELFORMAT** structure to the address passed in the second parameter (also a **DDPIXELFORMAT** structure), and returns D3DENUMRET_CANCEL to stop the enumeration.

After you determine the format of the depth buffer, you can create a DirectDrawSurface that uses that format, which is the topic of Step 2: Create the Depth Buffer.

Step 2: Create the Depth Buffer

[This is preliminary documentation and subject to change.]

Now that you have chosen the depth-buffer format, you can create the DirectDrawSurface object that will become the depth buffer. The pixel format of the surface is the one determined through enumeration, but the surface dimensions must be identical to the render-target surface to which it will be attached. The ZBuffer sample uses the following code for this task:

```
// If the enumerated format is good (it should be), the
// dwSize member will be properly initialized. Check this
// just in case.
if( sizeof(DDPIXELFORMAT) != ddpfZBuffer.dwSize )
    return E_FAIL;

// Get z-buffer dimensions from the render target
// Setup the surface desc for the z-buffer.
ddsd.dwFlags        = DDSD_CAPS|DDSD_WIDTH|DDSD_HEIGHT|DDSD_PIXELFORMAT;
ddsd.ddsCaps.dwCaps = DDSCAPS_ZBUFFER;
ddsd.dwWidth        = g_rcScreenRect.right - g_rcScreenRect.left;
ddsd.dwHeight       = g_rcScreenRect.bottom - g_rcScreenRect.top;
memcpy( &ddsd.ddpfPixelFormat, &ddpfZBuffer, sizeof(DDPIXELFORMAT) );

// Software devices require system-memory depth buffers.
if( IsEqualIID( *pDeviceGUID, IID_IDirect3DHALDevice ) )
    ddsd.ddsCaps.dwCaps |= DDSCAPS_VIDEOMEMORY;
else
    ddsd.ddsCaps.dwCaps |= DDSCAPS_SYSTEMMEMORY;

// Create the depth-buffer.
if( FAILED( hr = g_pDD4->CreateSurface( &ddsd, &g_pddsZBuffer, NULL ) ) )
    return hr;
```

The preceding code simply prepares a **DDSURFACEDESC2** structure for the depth buffer, using the dimensions of the render-target surface calculated from previously set global variables. The pixel format information retrieved during the previous step, Step 1: Enumerate Depth-Buffer Formats, is copied into the surface description.

Note

A hardware device can use a depth buffer regardless of its location in memory. When using a hardware device, it's best to let the device determine the best location for the buffer by omitting the DDSCAPS_VIDEOMEMORY and DDSCAPS_SYSTEMMEMORY surface capability flags. However, a software device can only be created if the depth buffer exists in system memory. The preceding code checks for this possibility and includes the DDSCAPS_SYSTEMMEMORY flag if necessary.

Once the surface description is ready, the code calls the **IDirectDraw4::CreateSurface** method to create the new depth-buffer surface. After the depth buffer is created, it can be attached to the surface that will be used as the render target, as described in Step 3: Attach the Depth Buffer.

Step 3: Attach the Depth Buffer

[This is preliminary documentation and subject to change.]

Once the depth buffer is created, you need to attach it to the surface that will be used as the render target. Do this by calling the **IDirectDrawSurface4::AddAttachedSurface** method of the render-target surface. The ZBuffer sample performs this with the following code:

```
// Attach the z-buffer to the back buffer.
if( FAILED( hr = g_pddsBackBuffer->AddAttachedSurface( g_pddsZBuffer ) ) )
    return hr;
```

Once the depth buffer is attached to the render-target surface, the system will automatically use the depth buffer whenever depth buffering is enabled, as discussed in Step 4: Enable Depth Buffering.

Step 4: Enable Depth Buffering

[This is preliminary documentation and subject to change.]

After attaching the depth buffer to the render-target surface, you can create a rendering device from the render target. Given a rendering device, you enable depth buffering by setting the D3DRENDERSTATE_ZENABLE render state for the device. The **D3DZBUFFERTYPE** enumerated type includes members to set the depth-buffer render state. The D3DZB_TRUE member (or TRUE) enables z-buffering. The ZBuffer sample enables z-buffering during scene rendering in the App_Render application-defined function. The following is the appropriate excerpt from App_Render:

```
// Enable z-buffering.
pd3dDevice->SetRenderState( D3DRENDERSTATE_ZENABLE, TRUE );
```

Although this tutorial enables depth-buffering each frame, it is not necessary to do so. A real-world application would likely set the D3DRENDERSTATE_ZENABLE render state during scene initialization, only changing to disable depth buffering or to choose another type of depth buffering.

Note

The **D3DZBUFFERTYPE** enumerated type includes the D3DZB_USEW value to enable w-based depth comparisons on compliant hardware. For more information, see Depth Buffers.

Direct3D Immediate Mode Visual Basic Tutorials

[This is preliminary documentation and subject to change.]

Tutorial 1: ?

[This is preliminary documentation and subject to change.]

Direct3D Immediate Mode Reference

[This is preliminary documentation and subject to change.]

This section contains reference information for the application programming interface (API) elements provided by Direct3D® Immediate Mode in C/C++ and Visual Basic. Reference material is organized by language:

- Direct3D Immediate Mode C/C++ Reference
- Direct3D Immediate Mode Visual Basic Reference

Direct3D Immediate Mode C/C++ Reference

[This is preliminary documentation and subject to change.]

This section contains reference information for the application programming interface (API) elements provided by Direct3D® Immediate Mode. Reference material is divided into the following categories:

- Interfaces
- D3D_OVERLOADS
- Callback Functions
- Macros
- Structures
- Enumerated Types
- Other Types

- Flexible Vertex Format Flags
- Return Values

Interfaces

[This is preliminary documentation and subject to change.]

This section contains reference information for the COM interfaces provided by Direct3D Immediate Mode. The following interfaces are covered:

- **IDirect3D3**
- **IDirect3DDevice**
- **IDirect3DDevice3**
- **IDirect3DExecuteBuffer**
- **IDirect3DLight**
- **IDirect3DMaterial3**
- **IDirect3DTexture2**
- **IDirect3DVertexBuffer**
- **IDirect3DViewport3**

Some stub methods that were exposed in previous releases of DirectX are no longer exposed. For more information, see Unimplemented Methods.

IDirect3D3

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirect3D3** interface to create Direct3D objects and set up the environment. This section is a reference to the methods of this interface. For a conceptual overview, see Direct3D Interfaces.

The **IDirect3D3** interface is obtained by calling the **QueryInterface** method from a DirectDraw object.

The methods of the **IDirect3D3** interface can be organized into the following groups:

Creation	**CreateDevice**
	CreateLight
	CreateMaterial
	CreateVertexBuffer
	CreateViewport
Enumeration	**EnumDevices**
	EnumZBufferFormats

	FindDevice
Miscellaneous	**EvictManagedTextures**

The **IDirect3D3** interface extends the **IDirect3D2** interface by adding methods that enable applications to create vertex buffers and enumerate texture map and depth-buffer formats.

The **IDirect3D3** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3D3, LPDIRECT3D2,** and **LPDIRECT3D** types are defined as pointers to the **IDirect3D3, IDirect3D2** and **IDirect3D** interfaces:

```
typedef struct IDirect3D    *LPDIRECT3D;
typedef struct IDirect3D2   *LPDIRECT3D2;
typedef struct IDirect3D3   *LPDIRECT3D3;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Accessing Direct3D, Direct3D and DirectDraw

IDirect3D3::CreateDevice

[This is preliminary documentation and subject to change.]

The **IDirect3D3::CreateDevice** method creates a Direct3D device to be used with the DrawPrimitive methods.

```
HRESULT CreateDevice(
  REFCLSID rclsid,
  LPDIRECTDRAWSURFACE4 lpDDS,
  LPDIRECT3DDEVICE3 *  lplpD3DDevice,
  LPUNKNOWN pUnkOuter   // See remarks
);
```

rclsid
Class identifier for the new device. This value can be IID_IDirect3DHALDevice, IID_IDirect3DMMXDevice, or IID_IDirect3DRGBDevice. The IID_IDirect3DRampDevice, used for the ramp emulation device, is not supported by **IDirect3D3**. To use ramp emulation, you must use the legacy **IDirect3D2** interface.

lpDDS
Address of the **IDirectDrawSurface4** interface for the DirectDrawSurface object that will be the device's rendering target. The surface must have been created as a 3-D device by using the DDSCAPS_3DDEVICE capability.

lplpD3DDevice
Address that points to the new **IDirect3DDevice3** interface when the method returns.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, the **IDirect3D3::CreateDevice** method returns an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3D2** interface. In previous versions of Direct3D, devices could be created only by calling the **IDirectDrawSurface::QueryInterface** method; devices created in this manner can only be used with execute buffers.

In the **IDirect3D2** interface, the **CreateDevice** method accepts only three parameters, which are identical to the first three parameters for the **IDirect3D3::CreateDevice** method.

When you call **IDirect3D3::CreateDevice**, you create a device object that is separate from a DirectDraw surface object. This device uses a DirectDraw surface as a rendering target.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3, Creating a Direct3D Device, Direct3D Devices

IDirect3D3::CreateLight

[This is preliminary documentation and subject to change.]

The **IDirect3D3::CreateLight** method allocates a Direct3DLight object. This object can then be associated with a viewport by using the **IDirect3DViewport3::AddLight** method.

```
HRESULT CreateLight(
  LPDIRECT3DLIGHT* lplpDirect3DLight,
  IUnknown* pUnkOuter
);
```

Parameters

lplpDirect3DLight
Address that will be filled with a pointer to an **IDirect3DLight** interface if the call succeeds.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, the **IDirect3D3::CreateLight** method returns an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3D2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DLight, Lights, Using Lights, Lighting and Materials

IDirect3D3::CreateMaterial

[This is preliminary documentation and subject to change.]

The **IDirect3D3::CreateMaterial** method allocates a Direct3DMaterial3 object.

```
HRESULT CreateMaterial(
  LPDIRECT3DMATERIAL3* lplpDirect3DMaterial,
  IUnknown* pUnkOuter
);
```

Parameters

lplpDirect3DMaterial
Address that will be filled with a pointer to an **IDirect3DMaterial3** interface if the call succeeds.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, **CreateMaterial** returns an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return codes, see Direct3D Immediate Mode Return Values.

Remarks

In the **IDirect3D2** interface, this method retrieves a pointer to an **IDirect3DMaterial2** interface, not an **IDirect3DMaterial3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Materials, Lighting and Materials

IDirect3D3::CreateVertexBuffer

[This is preliminary documentation and subject to change.]

The **IDirect3D3::CreateVertexBuffer** method creates a vertex buffer object.

```
HRESULT CreateVertexBuffer(
  LPD3DVERTEXBUFFERDESC lpVBDesc,
  LPDIRECT3DVERTEXBUFFER* lpD3DVertexBuffer,
  DWORD dwFlags,
  LPUNKNOWN pUnkOuter
);
```

Parameters

lpVBDesc
Address of a **D3DVERTEXBUFFERDESC** structure that describes the format and number of vertices that the vertex buffer will contain.

lpD3DVertexBuffer
Address of a variable that will contain a pointer to a **IDirect3DVertexBuffer** interface for the new vertex buffer.

dwFlags
Clipping flag value. Set this parameter to 0 to create a vertex buffer that can contain clipping information for untransformed or transformed vertices, or use the D3DDP_DONOTCLIP flag to create a vertex buffer that will contain transformed vertices, but no clipping information.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, **CreateVertexBuffer** returns an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

CLASS_E_NOAGGREGATION
D3DERR_INVALIDVERTEXFORMAT
D3DERR_VBUF_CREATE_FAILED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

Remarks

This method was introduced with the **IDirect3D3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DVertexBuffer, Vertex Buffers

IDirect3D3::CreateViewport

[This is preliminary documentation and subject to change.]

The **IDirect3D3::CreateViewport** method creates a Direct3DViewport object. The viewport is associated with a Direct3DDevice object by using the **IDirect3DDevice3::AddViewport** method.

```
HRESULT CreateViewport(
  LPDIRECT3DVIEWPORT3* lplpD3DViewport,
  IUnknown* pUnkOuter
);
```

Parameters

lplpD3DViewport
Address that will be filled with a pointer to an **IDirect3DViewport3** interface if the call succeeds.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, the **IDirect3D3::CreateViewport** method returns an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

In the **IDirect3D** interface, this method retrieves a pointer to an **IDirect3DViewport** interface, not an **IDirect3DViewport3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3D3::EnumDevices

[This is preliminary documentation and subject to change.]

The **IDirect3D3::EnumDevices** method enumerates all Direct3D device drivers installed on the system.

```
HRESULT EnumDevices(
  LPD3DENUMDEVICESCALLBACK lpEnumDevicesCallback,
  LPVOID lpUserArg
);
```

Parameters

lpEnumDevicesCallback
Address of the **D3DEnumDevicesCallback** callback function that the enumeration procedure will call every time a match is found.

lpUserArg
Address of application-defined data passed to the callback function.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

To use execute buffers with an MMX device, you must call the **IDirect3D3::CreateDevice** method to create an MMX **IDirect3DDevice3** interface and then use the **QueryInterface** method to create an **IDirect3DDevice** interface from **IDirect3DDevice3**.

MMX devices are enumerated only by **IDirect3D3::EnumDevices** and **IDirect3D2::EnumDevices**, not by **IDirect3D::EnumDevices**. If you use the **QueryInterface** method to create an **IDirect3D** interface from **IDirect3D3** before you enumerate the Direct3D drivers, the enumeration will behave like **IDirect3D::EnumDevices** — no MMX devices will be enumerated.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3D3::EnumZBufferFormats

[This is preliminary documentation and subject to change.]

The **IDirect3D3::EnumZBufferFormats** method enumerates the supported depth-buffer formats for a specified device.

```
HRESULT EnumZBufferFormats(
  REFCLSID riidDevice,
  LPD3DENUMPIXELFORMATSCALLBACK lpEnumCallback,
  LPVOID   lpContext
);
```

Parameters

riidDevice
Reference to a globally unique identifier for the device whose depth-buffer formats will be enumerated.

lpEnumCallback
Address of a **D3DEnumPixelFormatsCallback** callback function that will be called for each supported depth-buffer format.

lpContext
Application-defined data that is passed to the callback function.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOZBUFFERHW

DDERR_OUTOFMEMORY

Remarks

This method was introduced with the **IDirect3D3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3D3::EvictManagedTextures

[This is preliminary documentation and subject to change.]

The **IDirect3D3::EvictManagedTextures** method purges all managed textures from local or non-local video memory.

```
HRESULT EvictManagedTextures();
```

Parameters

None.

Return Values

This method returns D3D_OK.

Remarks

This method causes Direct3D to remove any texture surfaces created with the DDSCAPS2_TEXTUREMANAGE flag from local or non-local video memory.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Automatic Texture Management

IDirect3D3::FindDevice

[This is preliminary documentation and subject to change.]

The **IDirect3D3::FindDevice** method finds a device with specified characteristics and retrieves a description of it.

```
HRESULT FindDevice(
  LPD3DFINDDEVICESEARCH lpD3DFDS,
  LPD3DFINDDEVICERESULT lpD3DFDR
);
```

Parameters

lpD3DFDS
Address of the **D3DFINDDEVICESEARCH** structure describing the device to be located.

lpD3DFDR
Address of the **D3DFINDDEVICERESULT** structure describing the device if it is found.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return codes, see Direct3D Immediate Mode Return Values.

Remarks

This method is unchanged from its implementation in the **IDirect3D2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirect3DDevice** interface to retrieve and set the capabilities of Direct3D devices. This section is a reference to the methods of these interface. For a conceptual overview, see Direct3D Devices.

The **IDirect3DDevice** interface supports applications that work with execute buffers. It has been extended by the **IDirect3DDevice3** interface, which supports the DrawPrimitive methods.

The Direct3DDevice object is obtained by calling the **QueryInterface** method from a DirectDrawSurface object that was created as a 3-D–capable surface.

The methods of the **IDirect3DDevice** interface can be organized into the following groups.

Execute buffers	**CreateExecuteBuffer**
	Execute
Information	**EnumTextureFormats**
	GetCaps
	GetDirect3D
	GetPickRecords
	GetStats
Matrices	**CreateMatrix**
	DeleteMatrix
	GetMatrix
	SetMatrix
Miscellaneous	**Initialize**
	Pick
	SwapTextureHandles
Scenes	**BeginScene**
	EndScene
Viewports	**AddViewport**
	DeleteViewport
	NextViewport

The **IDirect3DDevice** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DDEVICE** type is defined as a pointer to the **IDirect3DDevice** interface:

```
typedef struct IDirect3DDevice          *LPDIRECT3DDEVICE;
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Execute Buffers

IDirect3DDevice::AddViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::AddViewport** method adds the specified viewport to the list of viewport objects associated with the device and increments the reference count of the viewport.

```
HRESULT AddViewport(
  LPDIRECT3DVIEWPORT lpDirect3DViewport
);
```

Parameters

lpDirect3DViewport
Address of the **IDirect3DViewport** interface of the viewport that should be associated with this device.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

In the **IDirect3DDevice3** interface, this method accepts a pointer to an **IDirect3DViewport3** interface.

This method will fail, returning DDERR_INVALIDPARAMS, if you attempt to add a viewport that has already been assigned to the device.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice::BeginScene

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::BeginScene** method begins a scene. Applications must call this method before performing any rendering, and must call **IDirect3DDevice::EndScene** when rendering is complete, and before calling **IDirect3DDevice::BeginScene** again.

```
HRESULT BeginScene();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

Remarks

Do not attempt to use GDI functions that use the device context of a render-target surface between calls to **BeginScene** and **EndScene**. Attempts to do so can prevent the results of the GDI operations from being visible. If your application uses GDI functions, make sure that all GDI calls are made outside of the scene functions.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice::EndScene, Beginning and Ending a Scene

IDirect3DDevice::CreateExecuteBuffer

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::CreateExecuteBuffer** method allocates an execute buffer for a display list.

```
HRESULT CreateExecuteBuffer(
  LPD3DEXECUTEBUFFERDESC   lpDesc,
  LPDIRECT3DEXECUTEBUFFER *lplpDirect3DExecuteBuffer,
  IUnknown *pUnkOuter
);
```

Parameters

lpDesc
Address of a **D3DEXECUTEBUFFERDESC** structure that describes the Direct3DExecuteBuffer object to be created. The call will fail if a buffer of at least the specified size cannot be created.

lplpDirect3DExecuteBuffer
Address of a pointer that will be filled with the address of the new Direct3DExecuteBuffer object.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, this method returns an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

The display list may be read by hardware DMA into VRAM for processing. All display primitives in the buffer that have indices to vertices must also have those vertices in the same buffer.

The **D3DEXECUTEBUFFERDESC** structure describes the execute buffer to be created. At a minimum, the application must specify the size required. If the

application specifies D3DDEBCAPS_VIDEOMEMORY in the **dwCaps** member, Direct3D will attempt to keep the execute buffer in video memory.

The application can use the **IDirect3DExecuteBuffer::Lock** method to request that the memory be moved. When this method returns, it will adjust the contents of the **D3DEXECUTEBUFFERDESC** structure to indicate whether the data resides in system or video memory.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice::CreateMatrix

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::CreateMatrix** method creates a matrix.

```
HRESULT CreateMatrix(
  LPD3DMATRIXHANDLE lpD3DMatHandle
);
```

Parameters

lpD3DMatHandle
Address of a variable that will contain a handle to the matrix that is created. The call will fail if a buffer of at least the size of the matrix cannot be created.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error, such as DDERR_INVALIDPARAMS.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice::DeleteMatrix, **IDirect3DDevice::SetMatrix**

IDirect3DDevice::DeleteMatrix

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::DeleteMatrix** method deletes a matrix handle. This matrix handle must have been created by using the **IDirect3DDevice::CreateMatrix** method.

```
HRESULT DeleteMatrix(
  D3DMATRIXHANDLE d3dMatHandle
);
```

Parameters

d3dMatHandle
Matrix handle to be deleted.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error, such as DDERR_INVALIDPARAMS.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice::CreateMatrix, **IDirect3DDevice::SetMatrix**

IDirect3DDevice::DeleteViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::DeleteViewport** method removes the specified viewport from the list of viewport objects associated with the device and decrements the reference count of the viewport.

```
HRESULT DeleteViewport(
  LPDIRECT3DVIEWPORT lpDirect3DViewport
```

```
);
```

Parameters

lpDirect3DViewport
Address of the **IDirect3DViewport** interface of the viewport object that will be disassociated with this device object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method fails, returning DDERR_INVALIDPARAMS, if you attempt to delete a viewport from the device without previously assigning the viewport with a call to **IDirect3DDevice::AddViewport**.

In the **IDirect3DDevice3** interface, this method accepts a pointer to an **IDirect3DViewport3** interface.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice::AddViewport

IDirect3DDevice::EndScene

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::EndScene** method ends a scene that was begun by calling the **IDirect3DDevice::BeginScene** method.

```
HRESULT EndScene();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

Remarks

When this method succeeds, the scene will have been rendered and the device surface will hold the contents of the rendering.

Do not attempt to use GDI functions that use the device context of a render-target surface between calls to **BeginScene** and **EndScene**. Attempts to do so can prevent the results of the GDI operations from being visible. If your application uses GDI functions, make sure that all GDI calls are made outside of the scene functions.

You must call this method before you can call the **IDirect3DDevice::BeginScene** method to start rendering another scene, even if the previous attempt to render was unsuccessful.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice::BeginScene, Beginning and Ending a Scene

IDirect3DDevice::EnumTextureFormats

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::EnumTextureFormats** method queries the current driver for a list of supported texture formats.

```
HRESULT EnumTextureFormats(
  LPD3DENUMTEXTUREFORMATSCALLBACK lpd3dEnumTextureProc,
  LPVOID lpArg
);
```

Parameters

lpd3dEnumTextureProc
Address of the **D3DEnumTextureFormatsCallback** function that the enumeration procedure will call for each texture format.

lpArg
Address of application-defined data passed to the callback function.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice::Execute

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::Execute** method executes a buffer.

```
HRESULT Execute(
  LPDIRECT3DEXECUTEBUFFER lpDirect3DExecuteBuffer,
  LPDIRECT3DVIEWPORT lpDirect3DViewport,
  DWORD dwFlags
);
```

Parameters

lpDirect3DExecuteBuffer
Address of the execute buffer to be executed.

lpDirect3DViewport
Address of the Direct3DViewport object that describes the transformation context into which the execute buffer will be rendered.

dwFlags
Flags specifying whether or not objects in the buffer should be clipped. This parameter must be one of the following values:

D3DEXECUTE_CLIPPED
Clip any primitives in the buffer that are outside or partially outside the viewport.

D3DEXECUTE_UNCLIPPED
All primitives in the buffer are contained within the viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

D3DEXECUTEDATA, **D3DINSTRUCTION**, **IDirect3DExecuteBuffer::Validate**

IDirect3DDevice::GetCaps

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::GetCaps** method retrieves the capabilities of the Direct3D device.

```
HRESULT GetCaps(
  LPD3DDEVICEDESC lpD3DHWDevDesc,
  LPD3DDEVICEDESC lpD3DHELDevDesc
);
```

Parameters

lpD3DHWDevDesc
Address of the **D3DDEVICEDESC** structure that will contain the hardware features of the device.

lpD3DHELDevDesc
Address of the **D3DDEVICEDESC** structure that will contain the software emulation being provided.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method does not retrieve the capabilities of the display device. To retrieve this information, use the **IDirectDraw4::GetCaps** method.

This method's implementation is unchanged in the **IDirect3DDevice3** interface.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice::GetDirect3D

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::GetDirect3D** method retrieves the Direct3D object for this device.

```
HRESULT GetDirect3D(
  LPDIRECT3D *lplpD3D
);
```

Parameters

lplpD3D
Address that will contain a pointer to the Direct3D object's **IDirect3D** interface when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return codes, see Direct3D Immediate Mode Return Values.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice::GetMatrix

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::GetMatrix** method retrieves a matrix from a matrix handle. This matrix handle must have been created by using the **IDirect3DDevice::CreateMatrix** method.

```
HRESULT GetMatrix(
  D3DMATRIXHANDLE D3DMatHandle,
  LPD3DMATRIX lpD3DMatrix
);
```

Parameters

D3DMatHandle
Handle to the matrix to be retrieved.

lpD3DMatrix
Address of a **D3DMATRIX** structure that contains the matrix when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error, such as DDERR_INVALIDPARAMS.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice::CreateMatrix, **IDirect3DDevice::DeleteMatrix**, **IDirect3DDevice::SetMatrix**

IDirect3DDevice::GetPickRecords

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::GetPickRecords** method retrieves the pick records for a device.

```
HRESULT GetPickRecords(
  LPDWORD lpCount,
  LPD3DPICKRECORD lpD3DPickRec
);
```

Parameters

lpCount
Address of a variable that contains the number of **D3DPICKRECORD** structures to retrieve.

lpD3DPickRec
Address that will contain an array of **D3DPICKRECORD** structures when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

Remarks

An application typically calls this method twice. In the first call, the second parameter is set to NULL, and the first parameter retrieves a count of all relevant **D3DPICKRECORD** structures. The application then allocates sufficient memory for those structures and calls the method again, specifying the newly allocated memory for the second parameter.

This method returns an unsorted list of pick records. Your application is responsible for sorting the records, if needed.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice::GetStats

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::GetStats** method retrieves statistics about a device.

```
HRESULT GetStats(
  LPD3DSTATS lpD3DStats
);
```

Parameters

lpD3DStats
Address of a **D3DSTATS** structure that will be filled with the statistics.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method can report inaccurately low statistics when used with DirectX 6.0 and later drivers. (The also applies to this method in the legacy **IDirect3DDevice2** interface.)

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice::Initialize

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::Initialize** method is not implemented.

```
HRESULT Initialize(
  LPDIRECT3D lpd3d,
  LPGUID lpGUID,
  LPD3DDEVICEDESC lpd3ddvdesc
```

```
);
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice::NextViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::NextViewport** method enumerates the viewports associated with the device.

```
HRESULT NextViewport(
  LPDIRECT3DVIEWPORT  lpDirect3DViewport,
  LPDIRECT3DVIEWPORT* lplpAnotherViewport,
  DWORD dwFlags
);
```

Parameters

lpDirect3DViewport
Address of the **IDirect3DViewport** interface of a viewport in the list of viewports associated with this Direct3D device.

lplpAnotherViewport
Address that will contain a pointer to the **IDirect3DViewport** interface for another viewport in the device's viewport list. Which viewport the method retrieves is determined by the flag in the *dwFlags* parameter.

dwFlags
Flag specifying which viewport to retrieve from the list of viewports. This must be set to one of the following flags:

D3DNEXT_HEAD
Retrieve the item at the beginning of the list.

D3DNEXT_NEXT
Retrieve the next item in the list.

D3DNEXT_TAIL
Retrieve the item at the end of the list.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

If you attempt to retrieve the next viewport in the list when you are at the end of the list, this method returns D3D_OK but *lplpAnotherViewport* is NULL.

In the **IDirect3DDevice3** interface, this method requires pointers to **IDirect3DViewport3** interfaces.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice::Pick

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::Pick** method executes a buffer without performing any rendering, but returns a z-ordered list of offsets to the primitives that intersect the upper-left corner of the rectangle specified by *lpRect*.

This call fails if the Direct3DExecuteBuffer object is locked.

```
HRESULT Pick(
  LPDIRECT3DEXECUTEBUFFER lpDirect3DExecuteBuffer,
  LPDIRECT3DVIEWPORT lpDirect3DViewport,
  DWORD dwFlags,
  LPD3DRECT lpRect
);
```

Parameters

lpDirect3DExecuteBuffer
Address of an execute buffer from which the z-ordered list is retrieved.

lpDirect3DViewport
Address of a viewport in the list of viewports associated with this Direct3DDevice object.

dwFlags
No flags are currently defined for this method.

lpRect

Address of a **D3DRECT** structure specifying the device coordinates to be picked. Currently, only primitives that intersect the **x1, y1** coordinates of this rectangle are returned. The **x2, y2** coordinates are ignored.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_LOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

The coordinates are specified in device-pixel space.

All Direct3DExecuteBuffer objects must be attached to a Direct3DDevice object in order for this method to succeed.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice::GetPickRecords

IDirect3DDevice::SetMatrix

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::SetMatrix** method applies a matrix to a matrix handle. This matrix handle must have been created by using the **IDirect3DDevice::CreateMatrix** method.

```
HRESULT SetMatrix(
  D3DMATRIXHANDLE d3dMatHandle,
  LPD3DMATRIX lpD3DMatrix
);
```

Parameters

d3dMatHandle
Matrix handle to be set.

lpD3DMatrix
Address of a **D3DMATRIX** structure that describes the matrix to be set.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error, such as DDERR_INVALIDPARAMS.

Remarks

Transformations inside the execute buffer include a handle to a matrix. The **IDirect3DDevice::SetMatrix** method enables an application to change this matrix without having to lock and unlock the execute buffer.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice::CreateMatrix, **IDirect3DDevice::GetMatrix**, **IDirect3DDevice::DeleteMatrix**

IDirect3DDevice::SwapTextureHandles

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice::SwapTextureHandles** method swaps two texture handles.

```
HRESULT SwapTextureHandles(
  LPDIRECT3DTEXTURE lpD3DTex1,
  LPDIRECT3DTEXTURE lpD3DTex2
);
```

Parameters

lpD3DTex1 and *lpD3DTex2*
Addresses of the **IDirect3DTexture** interfaces for the textures whose handles will be swapped.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

Remarks

This method is useful when an application is changing all the textures in a complicated object.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice3

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3** interface provides methods enabling applications to perform DrawPrimitive-based rendering; this is in contrast to the **IDirect3DDevice** interface, which applications use to work with execute buffers. You can create a Direct3DDevice object and retrieve a pointer to this interface by calling the **IDirect3D3::CreateDevice** method.

For a conceptual overview, see Direct3D Devices and The DrawPrimitive Methods.

The methods of the **IDirect3DDevice3** interface can be organized into the following groups:

Information	**GetCaps**
	GetDirect3D
	GetStats
Miscellaneous	**ComputeSphereVisibility**
	MultiplyTransform
Getting and Setting States	**GetClipStatus**
	GetCurrentViewport
	GetLightState
	GetRenderState

	GetRenderTarget
	GetTransform
	SetClipStatus
	SetCurrentViewport
	SetLightState
	SetRenderState
	SetRenderTarget
	SetTransform
Rendering	**Begin**
	BeginIndexed
	DrawIndexedPrimitive
	DrawIndexedPrimitiveStrided
	DrawIndexedPrimitiveVB
	DrawPrimitive
	DrawPrimitiveStrided
	DrawPrimitiveVB
	End
	Index
	Vertex
Scenes	**BeginScene**
	EndScene
Textures	**EnumTextureFormats**
	GetTexture
	GetTextureStageState
	SetTexture
	SetTextureStageState
	ValidateDevice
Viewports	**AddViewport**
	DeleteViewport
	NextViewport

The **IDirect3DDevice3** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**

QueryInterface
Release

This interface extends the **IDirect3DDevice2** interface by adding methods to support more flexible vertex formats, vertex buffers, and visibility computation. Note that all of the viewport-related methods in this interface accept slightly different parameters than their counterparts in the **IDirect3DDevice2** interface. Wherever an **IDirect3DDevice2** interface method might accept an **IDirect3DViewport2** interface pointer as a parameter, the methods in the **IDirect3DDevice3** interface accept an **IDirect3DViewport3** interface pointer instead.

This interface is not intended to be used with execute buffers, and therefore does not contain any execute-buffer related methods. If you need to use some of the methods in the **IDirect3DDevice** interface that are not supported in **IDirect3DDevice2** or **IDirect3DDevice3**, you can call **IDirect3DDevice2::QueryInterface** to retrieve a pointer to an **IDirect3DDevice** interface.

You can use the **LPDIRECT3DDEVICE2** or **LPDIRECT3DDEVICE3** data types to declare a variable that contains a pointer to an **IDirect3DDevice2** or **IDirect3DDevice3** interface. The D3d.h header file declares these data types with the following code:

```
typedef struct IDirect3DDevice2    *LPDIRECT3DDEVICE2;
typedef struct IDirect3DDevice3    *LPDIRECT3DDEVICE3;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Direct3D Devices, Rendering

IDirect3DDevice3::AddViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::AddViewport** method adds the specified viewport to the list of viewport objects associated with the device and increments the reference count of the viewport.

```
HRESULT AddViewport(
  LPDIRECT3DVIEWPORT3 lpDirect3DViewport
);
```

Parameters

lpDirect3DViewport
Address of the **IDirect3DViewport3** interface that should be associated with this Direct3DDevice object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method will fail, returning DDERR_INVALIDPARAMS, if you attempt to add a viewport that has already been assigned to the device.

In the **IDirect3DDevice2** interface, this method accepts a pointer to an **IDirect3DViewport** interface, not an **IDirect3DViewport3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice3::Begin

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::Begin** method indicates the start of a sequence of rendered primitives. This method defines the type of these primitives and the type of vertices on which they are based. The only method you can legally call between calls to **IDirect3DDevice3::Begin** and **IDirect3DDevice3::End** is **IDirect3DDevice3::Vertex**.

```
HRESULT Begin(
  D3DPRIMITIVETYPE d3dpt,
  DWORD dwVertexTypeDesc,
  DWORD dwFlags
);
```

Parameters

d3dpt

One of the members of the **D3DPRIMITIVETYPE** enumerated type.

dwVertexTypeDesc

A combination of flexible vertex format flags that describe the vertex format used. Only vertices that match this description will be accepted before the corresponding **IDirect3DDevice3::End**.

dwFlags

One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP

The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT

Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS

Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice3::GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method differs from its counterpart in the **IDirect3DDevice2** interface in that it accepts a flexible vertex format descriptor rather than a member of the **D3DVERTEXTYPE** enumerated type as the second parameter. If you attempt to use one of the members of **D3DVERTEXTYPE**, the method fails, returning DDERR_INVALIDPARAMS. For more information, see Vertex Formats.

This method fails if it is called after a call to the **IDirect3DDevice3::Begin** or **IDirect3DDevice3::BeginIndexed** method that has no bracketing call to **IDirect3DDevice3::End** method. Rendering calls that specify the wrong vertex type or that perform state changes will cause rendering of this primitive to fail.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::BeginIndexed, **IDirect3DDevice3::End**, **IDirect3DDevice3::Vertex**

IDirect3DDevice3::BeginIndexed

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::BeginIndexed** method defines the start of a primitive based on indexing into an array of vertices. This method fails if it is called after a call to the **IDirect3DDevice3::Begin** or **IDirect3DDevice3::BeginIndexed** method that has no corresponding call to **IDirect3DDevice3::End**. The only method you can legally call between calls to **IDirect3DDevice3::BeginIndexed** and **IDirect3DDevice3::End** is **IDirect3DDevice3::Index**.

```
HRESULT BeginIndexed(
  D3DPRIMITIVETYPE dptPrimitiveType,
  DWORD  dwVertexTypeDesc,
  LPVOID lpvVertices,
  DWORD  dwNumVertices,
  DWORD  dwFlags
);
```

Parameters

dptPrimitiveType
Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type. Note that the D3DPT_POINTLIST member of **D3DPRIMITIVETYPE** is not indexed.

dwVertexTypeDesc
A combination of flexible vertex format flags that describes the vertex format being used. Only vertices that match this type will be accepted before the corresponding **IDirect3DDevice3::End**.

lpvVertices
Pointer to the list of vertices to be used in the primitive sequence.

dwNumVertices
Number of vertices in the array at *lpvVertices*.

dwFlags
One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP
The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT
Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS
Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice3::GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

This method differs from its counterpart in the **IDirect3DDevice2** interface in that it accepts a flexible vertex format descriptor rather than a member of the **D3DVERTEXTYPE** enumerated type as the second parameter. If you attempt to use

one of the members of **D3DVERTEXTYPE**, the method fails, returning DDERR_INVALIDPARAMS. For more information, see Vertex Formats.

This method was first introduced in the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::Begin, **IDirect3DDevice3::End**, **IDirect3DDevice3::Index**

IDirect3DDevice3::BeginScene

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::BeginScene** method begins a scene. Applications must call this method before performing any rendering, and must call **IDirect3DDevice3::EndScene** when rendering is complete, and before calling **IDirect3DDevice::BeginScene** again.

```
HRESULT BeginScene();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::EndScene

IDirect3DDevice3::ComputeSphereVisibility

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::ComputeSphereVisibility** method calculates the visibility (complete, partial, or no visibility) of an array spheres within the current viewport for this device.

```
HRESULT ComputeSphereVisibility(
  LPD3DVECTOR lpCenters,
  LPD3DVALUE  lpRadii,
  DWORD       dwNumSpheres,
  DWORD       dwFlags,
  LPDWORD     lpdwReturnValues
);
```

Parameters

lpCenters
Array of **D3DVECTOR** structures describing the center point for each sphere, in world-space coordinates.

lpRadii
Array of **D3DVALUE** variables that represent the radius for each sphere.

dwNumSpheres
Number of spheres. This value must be greater than zero.

dwFlags
Not currently used; set to zero.

lpdwReturnValues
Array of **DWORD** values. The array need not be initialized, but it must be large enough to contain a **DWORD** for each sphere being tested. When the method returns, each element in the array contains a combination of the following flags that describe the visibility of that sphere within the current viewport for this device:

Inside flags
D3DVIS_INSIDE_BOTTOM, D3DVIS_INSIDE_FAR, D3DVIS_INSIDE_FRUSTUM, D3DVIS_INSIDE_LEFT D3DVIS_INSIDE_NEAR, D3DVIS_INSIDE_RIGHT, D3DVIS_INSIDE_TOP
The sphere is inside the viewing frustum of the current viewport.

Intersection flags
D3DVIS_INTERSECT_BOTTOM or D3DVIS_INTERSECT_TOP
The sphere intersects the bottom or top plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_INTERSECT_FAR or D3DVIS_INTERSECT_NEAR

The sphere intersects the far or near plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_INTERSECT_FRUSTUM
The sphere intersects some part of the viewing frustum for the current viewport.

D3DVIS_INTERSECT_LEFT or D3DVIS_INTERSECT_RIGHT
The sphere intersects the left or right plane of the viewing frustum for the current viewport, depending on which flag is present.

Outside flags

D3DVIS_OUTSIDE_BOTTOM or D3DVIS_OUTSIDE_TOP
The sphere is outside the bottom or top plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_OUTSIDE_FAR or D3DVIS_OUTSIDE_NEAR
The sphere is outside the far or near plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_OUTSIDE_FRUSTUM
The sphere is somewhere outside the viewing frustum for the current viewport.

D3DVIS_OUTSIDE_LEFT or D3DVIS_OUTSIDE_RIGHT
The sphere is outside the left or right plane of the viewing frustum for the current viewport, depending on which flag is present.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDMATRIX

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

Sphere visibility is computed by back transforming the viewing frustum to the model space, using the inverse of the combined world, view or projection matrices. If the combined matrix can not be inverted (if the determinant is zero), the method fails, returning D3DERR_INVALIDMATRIX.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice3::DeleteViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::DeleteViewport** method removes the specified viewport from the list of viewport objects associated with the device and decrements the reference count of the viewport.

```
HRESULT DeleteViewport(
  LPDIRECT3DVIEWPORT3 lpDirect3DViewport
);
```

Parameters

lpDirect3DViewport
Address of the **IDirect3DViewport3** interface of the viewport object that will be disassociated with this device.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method fails, returning DDERR_INVALIDPARAMS, if you attempt to delete a viewport from the device without previously assigning the viewport with a call to **IDirect3DDevice3::AddViewport**.

In the **IDirect3DDevice2** interface, this method accepts a pointer to an **IDirect3DViewport** interface, not an **IDirect3DViewport3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::AddViewport

IDirect3DDevice3::DrawIndexedPrimitive

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::DrawIndexedPrimitive** method renders the specified geometric primitive based on indexing into an array of vertices.

```
HRESULT DrawIndexedPrimitive(
  D3DPRIMITIVETYPE d3dptPrimitiveType,
  DWORD  dwVertexTypeDesc,
  LPVOID lpvVertices,
  DWORD  dwVertexCount,
  LPWORD lpwIndices,
  DWORD  dwIndexCount,
  DWORD  dwFlags
);
```

Parameters

d3dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

Note that the D3DPT_POINTLIST member of **D3DPRIMITIVETYPE** is not indexed.

dwVertexTypeDesc

A combination of flexible vertex format flags that describes the vertex format for this set of primitives.

lpvVertices

Pointer to the list of vertices to be used in the primitive sequence.

dwVertexCount

Defines the number of vertices in the list.

Notice that this parameter is used differently from the *dwVertexCount* parameter in the **IDirect3DDevice3::DrawPrimitive** method. In that method, the *dwVertexCount* parameter gives the number of vertices to draw, but here it gives the total number of vertices in the array pointed to by the *lpvVertices* parameter. When you call **IDirect3DDevice3::DrawIndexedPrimitive**, you specify the number of vertices to draw in the *dwIndexCount* parameter.

lpwIndices

Pointer to a list of **WORD**s that are to be used to index into the specified vertex list when creating the geometry to render.

dwIndexCount

Specifies the number of indices provided for creating the geometry. The maximum number of indices allowed is 65,535 (0xFFFF).

dwFlags

One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP
The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT
Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS
Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice3::GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE
D3DERR_INVALIDPRIMITIVETYPE
D3DERR_INVALIDVERTEXTYPE
DDERR_INVALIDPARAMS
DDERR_WASSTILLDRAWING

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

Do not use this method to render very small subsets of vertices from extremely large vertex arrays. This method transforms every vertex in the provided buffer, regardless of the location or quantity of vertices being rendered. Thus, if you pass an array that contains thousands of vertices, but only intend to render hundreds, your application's performance will suffer dramatically. In cases where you need to render a small

number of vertices from a large buffer, use the Direct3D vertex buffer rendering methods. For more information, see Vertex Buffers.

This method differs from its counterpart in the **IDirect3DDevice2** interface in that it accepts a flexible vertex format descriptor rather than a member of the **D3DVERTEXTYPE** enumerated type as the second parameter. If you attempt to use one of the members of **D3DVERTEXTYPE**, the method fails, returning DDERR_INVALIDPARAMS. For more information, see Vertex Formats.

In current versions of DirectX, **IDirect3DDevice3::DrawIndexedPrimitive** can sometimes generate an update rectangle that is larger than it strictly needs to be. If a large number of vertices need to be processed, this can have a negative impact on the performance of your application. If you are using **D3DTLVERTEX** vertices and the system is processing more vertices than you need, you should use the D3DDP_DONOTCLIP and D3DDP_DONOTUPDATEEXTENTS flags to solve the problem.

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::DrawPrimitive, **IDirect3DDevice3::DrawPrimitiveStrided**, **IDirect3DDevice3::DrawPrimitiveVB**, **IDirect3DDevice3::DrawIndexedPrimitiveStrided**, **IDirect3DDevice3::DrawIndexedPrimitiveVB**

IDirect3DDevice3::DrawIndexedPrimitive Strided

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::DrawIndexedPrimitiveStrided** method renders a geometric primitive based on indexing into an array of strided vertices. For more information, see Strided Vertex Format.

HRESULT DrawIndexedPrimitiveStrided(
D3DPRIMITIVETYPE *d3dptPrimitiveType*,
DWORD *dwVertexTypeDesc*,
LPD3DDRAWPRIMITIVESTRIDEDDATA *lpVertexArray*,
DWORD *dwVertexCount*,
LPWORD *lpwIndices*,

```
  DWORD  dwIndexCount,
  DWORD  dwFlags
);
```

Parameters

d3dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

Note that the D3DPT_POINTLIST member of **D3DPRIMITIVETYPE** is not indexed.

dwVertexTypeDesc

A combination of flexible vertex format flags vertex format for this primitive.

lpVertexArray

Array of **D3DDRAWPRIMITIVESTRIDEDDATA** structures that contains the vertices for this primitive, in the format specified by the flags in *dwVertexTypeDesc*.

dwVertexCount

Defines the number of vertices in the list.

Notice that this parameter is used differently from the *dwVertexCount* parameter in the **IDirect3DDevice3::DrawPrimitive** method. In that method, the *dwVertexCount* parameter gives the number of vertices to draw, but here it gives the total number of vertices in the array pointed to by the *lpVertexArray* parameter. When you call **IDirect3DDevice3::DrawIndexedPrimitiveStrided**, you specify the number of vertices to draw in the *dwIndexCount* parameter.

lpwIndices

Pointer to a list of **WORD**s that are to be used to index into the specified vertex list when creating the geometry to render.

dwIndexCount

Specifies the number of indices provided for creating the geometry. The maximum number of indices allowed is 65,535 (0xFFFF).

dwFlags

One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP

The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT

Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS

Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice3::GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE
D3DERR_INVALIDPRIMITIVETYPE
D3DERR_INVALIDVERTEXTYPE
DDERR_INVALIDPARAMS
DDERR_WASSTILLDRAWING

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

This method does not support transformed vertices. As a result, if you include the D3DFVF_XYZRHW vertex format descriptor in the *dwVertexTypeDesc* parameter, the method fails, returning D3DERR_INVALIDVERTEXTYPE.

This method was introduced with the **IDirect3DDevice3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::DrawPrimitive, **IDirect3DDevice3::DrawPrimitiveStrided**, **IDirect3DDevice3::DrawPrimitiveVB**, **IDirect3DDevice3::DrawIndexedPrimitive**, **IDirect3DDevice3::DrawIndexedPrimitiveVB**, Strided Vertex Format

IDirect3DDevice3::DrawIndexedPrimitiveVB

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::DrawIndexedPrimitiveVB** method renders a geometric primitive based on indexing into an array of vertices within a vertex buffer.

```
HRESULT DrawIndexedPrimitiveVB(
  D3DPRIMITIVETYPE d3dptPrimitiveType,
  LPDIRECT3DVERTEXBUFFER lpd3dVertexBuffer,
  LPWORD lpwIndices,
  DWORD  dwIndexCount,
  DWORD  dwFlags
);
```

Parameters

d3dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

Note that the D3DPT_POINTLIST member of **D3DPRIMITIVETYPE** is not indexed.

lpd3dVertexBuffer

Address of the **IDirect3DVertexBuffer** interface for the vertex buffer that contains the array of vertices. Vertices can be transformed or untransformed, optimized or unoptimized.

lpwIndices

Address of an array of **WORD**s that will be used to index into the vertices in the vertex buffer.

dwIndexCount

The number of indices in the array at *lpwIndices*. The maximum number of indices allowed is 65,535 (0xFFFF).

dwFlags

One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP

The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT
Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS
Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice3::GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDPRIMITIVETYPE
D3DERR_INVALIDVERTEXTYPE
D3DERR_VERTEXBUFFERLOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_WASSTILLDRAWING

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

Software devices—MMX and RGB devices—cannot render from a video memory (local or non-local) vertex buffer. To render a vertex buffer using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

You cannot render from a locked vertex buffer; calls to the **IDirect3DDevice3::DrawIndexedPrimitiveVB** or **IDirect3DDevice3::DrawPrimitiveVB** method using a locked buffer will fail, returning D3DERR_VERTEXBUFFERLOCKED.

This method was introduced with the **IDirect3DDevice3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::DrawPrimitive, **IDirect3DDevice3::DrawPrimitiveStrided**, **IDirect3DDevice3::DrawPrimitiveVB**, **IDirect3DDevice3::DrawIndexedPrimitive**, **IDirect3DDevice3::DrawIndexedPrimitiveStrided**

IDirect3DDevice3::DrawPrimitive

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::DrawPrimitive** method renders the specified array of vertices as a sequence of geometric primitives of the specified type.

```
HRESULT DrawPrimitive(
  D3DPRIMITIVETYPE dptPrimitiveType,
  DWORD  dwVertexTypeDesc,
  LPVOID lpvVertices,
  DWORD  dwVertexCount,
  DWORD  dwFlags
);
```

Parameters

dptPrimitiveType
Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

dwVertexTypeDesc
A combination of flexible vertex format flags that describe the vertex format used for this set of primitives.

lpvVertices
Pointer to the array of vertices to be used in the primitive sequence.

dwVertexCount

The number of vertices in the array. The maximum number of vertices allowed is 65,535 (0xFFFF).

dwFlags

One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP

The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT

Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS

Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice3::GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE

D3DERR_INVALIDPRIMITIVETYPE

D3DERR_INVALIDVERTEXTYPE

DDERR_INVALIDPARAMS

DDERR_WASSTILLDRAWING

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

This method differs from its counterpart in the **IDirect3DDevice2** interface in that it accepts a flexible vertex format descriptor rather than a member of the **D3DVERTEXTYPE** enumerated type as the second parameter. If you attempt to use one of the members of **D3DVERTEXTYPE**, the method fails, returning DDERR_INVALIDPARAMS. For more information, see Vertex Formats.

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::DrawPrimitiveStrided,
IDirect3DDevice3::DrawPrimitiveVB,
IDirect3DDevice3::DrawIndexedPrimitive,
IDirect3DDevice3::DrawIndexedPrimitiveStrided,
IDirect3DDevice3::DrawIndexedPrimitiveVB

IDirect3DDevice3::DrawPrimitiveStrided

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::DrawPrimitiveStrided** method renders the specified array of strided vertices as a sequence of geometric primitives. For more information, see Strided Vertex Format.

```
HRESULT DrawPrimitiveStrided(
  D3DPRIMITIVETYPE dptPrimitiveType,
  DWORD  dwVertexTypeDesc,
  LPD3DDRAWPRIMITIVESTRIDEDDATA lpVertexArray,
  DWORD  dwVertexCount,
  DWORD  dwFlags
);
```

Parameters

dptPrimitiveType
Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

dwVertexTypeDesc
A combination of flexible vertex format flags that describe the vertex format.

lpVertexArray

Array of **D3DDRAWPRIMITIVESTRIDEDDATA** structures that contains the vertices for this primitive.

dwVertexCount

Number of vertices in the array at *lpVertexArray*. The maximum number of vertices allowed is 65,535 (0xFFFF).

dwFlags

One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP

The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT

Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS

Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice3::GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE

D3DERR_INVALIDPRIMITIVETYPE

D3DERR_INVALIDVERTEXTYPE

DDERR_INVALIDPARAMS

DDERR_WASSTILLDRAWING

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride

match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

This method does not support transformed vertices. As a result, if you include the D3DFVF_XYZRHW vertex format descriptor in the *dwVertexTypeDesc* parameter, the method fails, returning D3DERR_INVALIDVERTEXTYPE.

This method was introduced with the **IDirect3DDevice3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::DrawPrimitive, **IDirect3DDevice3::DrawPrimitiveVB**, **IDirect3DDevice3::DrawIndexedPrimitive**, **IDirect3DDevice3::DrawIndexedPrimitiveStrided**, **IDirect3DDevice3::DrawIndexedPrimitiveVB**, Strided Vertex Format

IDirect3DDevice3::DrawPrimitiveVB

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::DrawPrimitiveVB** method renders an array of vertices in a vertex buffer as a sequence of geometric primitives.

```
HRESULT DrawPrimitiveVB(
  D3DPRIMITIVETYPE d3dptPrimitiveType,
  LPDIRECT3DVERTEXBUFFER lpd3dVertexBuffer,
  DWORD dwStartVertex,
  DWORD dwNumVertices,
  DWORD dwFlags
);
```

Parameters

d3dptPrimitiveType
Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

lpd3dVertexBuffer
Address of the **IDirect3DVertexBuffer** interface for the vertex buffer that contains the array of vertices. Vertices can be transformed or untransformed, optimized or unoptimized.

dwStartVertex

Index value of the first vertex in the primitive. The highest possible starting index is 65,535 (0xFFFF). In debug builds, specifying a starting index value that exceeds this limit will cause the method to fail and return DDERR_INVALIDPARAMS.

dwNumVertices

Number of vertices to be rendered. The maximum number of vertices allowed is 65,535 (0xFFFF).

dwFlags

One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP

The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT

Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS

Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice3::GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT

Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE

D3DERR_INVALIDPRIMITIVETYPE

D3DERR_INVALIDVERTEXTYPE

D3DERR_VERTEXBUFFERLOCKED

DDERR_INVALIDPARAMS

DDERR_WASSTILLDRAWING

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

Software devices—MMX and RGB devices—cannot render from a video memory (local or non-local) vertex buffer. To render a vertex buffer using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

You cannot render from a locked vertex buffer; calls to the **IDirect3DDevice3::DrawIndexedPrimitiveVB** or **IDirect3DDevice3::DrawPrimitiveVB** method using a locked buffer will fail, returning D3DERR_VERTEXBUFFERLOCKED.

This method was introduced with the **IDirect3DDevice3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::DrawPrimitive, **IDirect3DDevice3::DrawPrimitiveStrided**, **IDirect3DDevice3::DrawIndexedPrimitive**, **IDirect3DDevice3::DrawIndexedPrimitiveStrided**, **IDirect3DDevice3::DrawIndexedPrimitiveVB**

IDirect3DDevice3::End

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::End** method signals the completion of a primitive sequence. This method fails if no corresponding call to the **IDirect3DDevice3::Begin** method (or **IDirect3DDevice3::BeginIndexed**) was made.

```
HRESULT End(
  DWORD dwFlags
);
```

Parameters

dwFlags

Reserved for future use; set to zero.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

This method fails if the vertex count is incorrect for the primitive type. It fails without drawing if it is called before a sufficient number of vertices is specified. If the number of **IDirect3DDevice3::Vertex** or **IDirect3DDevice3::Index** calls made is not evenly divisible by 3 (in the case of triangles), or 2 (in the case of a line list), the remainder will be ignored.

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::Begin, **IDirect3DDevice3::BeginIndexed**

IDirect3DDevice3::EndScene

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::EndScene** method ends a scene that was begun by calling the **IDirect3DDevice3::BeginScene** method.

```
HRESULT EndScene();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

Remarks

When this method succeeds, the scene will have been rendered and the device surface will hold the contents of the rendering.

You must call this method before you can call the **IDirect3DDevice3::BeginScene** method to start rendering another scene, even if the previous attempt to render was unsuccessful.

This method is unchanged from its implementation in the **IDirect3DDevice** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::BeginScene

IDirect3DDevice3::EnumTextureFormats

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::EnumTextureFormats** method queries the current driver for a list of supported texture formats.

```
HRESULT EnumTextureFormats(
  LPD3DENUMPIXELFORMATSCALLBACK lpd3dEnumPixelProc,
  LPVOID lpArg
);
```

Parameters

lpd3dEnumTextureProc
Address of the **D3DEnumPixelFormatsCallback** callback function that the enumeration procedure will call for each texture format.

lpArg
Address of application-defined data passed to the callback function.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

In the **IDirect3DDevice2** interface, this method accepts a pointer to a **D3DEnumTextureFormatsCallback** function, not a **D3DEnumPixelFormatsCallback**.

This method might not enumerate newly implemented texture formats on some devices. Applications that require a texture format that isn't enumerated can attempt to create a surface of that format. If the creation attempt succeeds, the format is supported.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice3::GetCaps

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetCaps** method retrieves the capabilities of the Direct3D device.

```
HRESULT GetCaps(
  LPD3DDEVICEDESC lpD3DHWDevDesc,
  LPD3DDEVICEDESC lpD3DHELDevDesc
);
```

Parameters

lpD3DHWDevDesc
Address of the **D3DDEVICEDESC** structure that will contain the hardware features of the device.

lpD3DHELDevDesc
Address of the **D3DDEVICEDESC** structure that will contain the software emulation being provided.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method does not retrieve the capabilities of the display device. To retrieve this information, use the **IDirectDraw4::GetCaps** method.

This method is unchanged from its implementation in the **IDirect3DDevice** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice3::GetClipStatus

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetClipStatus** method gets the current clip status.

```
HRESULT GetClipStatus(
  LPD3DCLIPSTATUS lpD3DClipStatus
);
```

Parameters

lpD3DClipStatus
Address of a **D3DCLIPSTATUS** structure that describes the current clip status.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::SetClipStatus

IDirect3DDevice3::GetCurrentViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetCurrentViewport** method retrieves the current viewport.

```
HRESULT GetCurrentViewport(
  LPDIRECT3DVIEWPORT3 *lplpd3dViewport
);
```

Parameters

lplpd3dViewport
Address that will contain a pointer to the current viewport's **IDirect3DViewport3** interface when the method returns. A reference is taken to the viewport object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDPARAMS	One of the arguments is invalid.
D3DERR_NOCURRENTVIEWPORT	No current viewport has been set by a call to the **IDirect3DDevice3::SetCurrentViewport** method.

Remarks

This method increases the reference count of the viewport interface retrieved in the *lplpd3dViewport* parameter. The application must release this interface when it is no longer needed.

This method was introduced with the **IDirect3DDevice2** interface. In the **IDirect3DDevice2** interface, this method accepts a pointer to an **IDirect3DViewport** interface, not an **IDirect3DViewport3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::SetCurrentViewport

IDirect3DDevice3::GetDirect3D

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetDirect3D** method retrieves the Direct3D object for this device.

```
HRESULT GetDirect3D(
  LPDIRECT3D3 *lplpD3D
);
```

Parameters

lplpD3D
Address that will contain a pointer to the Direct3D object's **IDirect3D3** interface when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return codes, see Direct3D Immediate Mode Return Values.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice3::GetLightState

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetLightState** method gets a single Direct3D device lighting-related state value.

```
HRESULT GetLightState(
  D3DLIGHTSTATETYPE dwLightStateType,
  LPDWORD lpdwLightState
);
```

Parameters

dwLightStateType
Device state variable that is being queried. This parameter can be any of the members of the **D3DLIGHTSTATETYPE** enumerated type.

lpdwLightState
Address of a variable that will contain the Direct3DDevice light state when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::SetLightState

IDirect3DDevice3::GetRenderState

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetRenderState** method gets a single Direct3DDevice rendering state parameter.

```
HRESULT GetRenderState(
  D3DRENDERSTATETYPE dwRenderStateType,
  LPDWORD lpdwRenderState
);
```

Parameters

dwRenderStateType
Device state variable that is being queried. This parameter can be any of the members of the **D3DRENDERSTATETYPE** enumerated type.

lpdwRenderState
Address of a variable that will contain the Direct3DDevice render state when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::SetRenderState

IDirect3DDevice3::GetRenderTarget

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetRenderTarget** method retrieves a pointer to the DirectDraw surface that is being used as a render target.

```
HRESULT GetRenderTarget(
  LPDIRECTDRAWSURFACE4 *lplpRenderTarget
);
```

Parameters

lplpRenderTarget
Address that will contain a pointer to the **IDirectDrawSurface4** interface of the render target surface for this device.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::SetRenderTarget

IDirect3DDevice3::GetStats

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetStats** method is obsolete, and not implemented in the **IDirect3DDevice3** interface.

```
HRESULT GetStats(
  LPD3DSTATS lpD3DStats
```

```
);
```

Return Values

The method returns E_NOTIMPL.

Remarks

This method is implemented in the **IDirect3DDevice2** and **IDirect3DDevice** interfaces, but can under-report statistics when used with DirectX 6.0 and later drivers.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice3::GetTexture

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetTexture** method retrieves a texture assigned to a given stage for a device.

```
HRESULT  GetTexture(
  DWORD dwStage,
  LPDIRECT3DTEXTURE2 * lplpTexture
);
```

Parameters

dwStage
Stage identifier of the texture to be retrieved. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value allowed for *dwStage* is 7.

lplpTexture
Address of a variable that will be filled with a pointer to the specified texture's **IDirect3DTexture2** interface if the call succeeds.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method was introduced with the **IDirect3DDevice3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::SetTexture, **IDirect3DDevice3::GetTextureStageState**, **IDirect3DDevice3::SetTextureStageState**, Textures

IDirect3DDevice3::GetTextureStageState

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetTextureStageState** method retrieves a state value for a currently assigned texture.

```
HRESULT GetTextureStageState(
  DWORD   dwStage,
  D3DTEXTURESTAGESTATETYPE dwState,
  LPDWORD lpdwValue
);
```

Parameters

dwStage
Stage identifier of the texture for which the state will be retrieved. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value allowed for *dwStage* is 7.

dwState
Texture state to be retrieved. This parameter can be any member of the **D3DTEXTURESTAGESTATETYPE** enumerated type.

lpdwValue
Address of a variable that will be filled with the retrieved state value. The meaning of the retrieved value is determined by the *dwState* parameter.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method was introduced with the **IDirect3DDevice3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::SetTextureStageState, **IDirect3DDevice3::GetTexture**, **IDirect3DDevice3::SetTexture**, Textures

IDirect3DDevice3::GetTransform

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::GetTransform** method gets a matrix describing a transformation state.

```
HRESULT GetTransform(
  D3DTRANSFORMSTATETYPE dtstTransformStateType,
  LPD3DMATRIX lpD3DMatrix
);
```

Parameters

dtstTransformStateType
Device state variable that is being modified. This parameter can be any of the members of the **D3DTRANSFORMSTATETYPE** enumerated type.

lpD3DMatrix
Address of a **D3DMATRIX** structure describing the transformation.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::SetTransform

IDirect3DDevice3::Index

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::Index** method adds a new index to the primitive sequence started with a previous call to the **IDirect3DDevice3::BeginIndexed** method.

```
HRESULT Index(
  WORD wVertexIndex
);
```

Parameters

wVertexIndex
Index of the next vertex to be added to the currently started primitive sequence.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::BeginIndexed, **IDirect3DDevice3::End**

IDirect3DDevice3::MultiplyTransform

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::MultiplyTransform** method multiplies a device's world, view, or projection matrices by a specified matrix. The multiplication order is *lpD3DMatrix* times *dtstTransformStateType*.

```
HRESULT MultiplyTransform(
  D3DTRANSFORMSTATETYPE dtstTransformStateType,
  LPD3DMATRIX lpD3DMatrix
);
```

Parameters

dtstTransformStateType
A member of the **D3DTRANSFORMSTATETYPE** enumerated type that identifies which device matrix is to be modified. The most common setting, D3DTRANSFORMSTATE_WORLD, modifies the world matrix, but you can specify that the method modify the view or projection matrices if needed.

lpD3DMatrix
Address of a **D3DMATRIX** structure that modifies the current transformation.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

An application might use the **MultiplyTransform** method to work with hierarchies of transformations. For example, the geometry and transformations describing an arm might be arranged in the following hierarchy:

```
shoulder_transformation
   upper_arm geometry
   elbow transformation
      lower_arm geometry
      wrist transformation
         hand geometry
```

An application might use the following series of calls to render this hierarchy. (Not all of the parameters are shown in this pseudocode.)

```
IDirect3DDevice3::SetTransform(D3DTRANSFORMSTATE_WORLD,
   shoulder_transform)
IDirect3DDevice3::DrawPrimitive(upper_arm)
IDirect3DDevice3::MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
   elbow_transform)
IDirect3DDevice3::DrawPrimitive(lower_arm)
IDirect3DDevice3::MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
   wrist_transform)
IDirect3DDevice3::DrawPrimitive(hand)
```

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::DrawPrimitive, **IDirect3DDevice3::SetTransform**

IDirect3DDevice3::NextViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::NextViewport** method enumerates the viewports associated with the device.

```
HRESULT NextViewport(
  LPDIRECT3DVIEWPORT3 lpDirect3DViewport,
```

```
  LPDIRECT3DVIEWPORT3 *lplpAnotherViewport,
  DWORD dwFlags
);
```

Parameters

lpDirect3DViewport
Address of the **IDirect3DViewport3** interface of a viewport in the list of viewports associated with this Direct3D device.

lplpAnotherViewport
Address that will contain a pointer to the **IDirect3DViewport3** interface for another viewport in the device's viewport list. Which viewport the method retrieves is determined by the flag in the *dwFlags* parameter.

dwFlags
Flag specifying which viewport to retrieve from the list of viewports. This must be set to one of the following flags:

D3DNEXT_HEAD
Retrieve the item at the beginning of the list.

D3DNEXT_NEXT
Retrieve the next item in the list.

D3DNEXT_TAIL
Retrieve the item at the end of the list.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

If you attempt to retrieve the next viewport in the list when you are at the end of the list, this method returns D3D_OK but *lplpAnotherViewport* is NULL.

In the **IDirect3DDevice2** interface, this method requires pointers to **IDirect3DViewport2** interfaces, not **IDirect3DViewport3** interfaces.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DDevice3::SetClipStatus

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::SetClipStatus** method sets the current clip status.

```
HRESULT SetClipStatus(
  LPD3DCLIPSTATUS lpD3DClipStatus
);
```

Parameters

lpD3DClipStatus
Address of a **D3DCLIPSTATUS** structure that describes the new settings for the clip status.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::GetClipStatus

IDirect3DDevice3::SetCurrentViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::SetCurrentViewport** method sets the current viewport.

```
HRESULT SetCurrentViewport(
```

LPDIRECT3DVIEWPORT3 *lpd3dViewport*
);

Parameters

lpd3dViewport
Address of the **IDirect3DViewport3** interface for the viewport that will become the current viewport if the method is successful.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

Applications must call this method before calling any rendering functions. Before calling this method, applications must have already called the **IDirect3DDevice3::AddViewport** method to add the viewport to the device.

Before the first call to **IDirect3DDevice3::SetCurrentViewport**, the current viewport for the device is invalid, and any attempts to render using the device will fail.

This method increases the reference count of the viewport interface specified by the *lpd3dViewport* parameter and releases the previous viewport, if any.

This method was introduced with the **IDirect3DDevice2** interface. In the **IDirect3DDevice2** interface, this method accepts a pointer to an **IDirect3DViewport** interface, not an **IDirect3DViewport3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::GetCurrentViewport

IDirect3DDevice3::SetLightState

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::SetLightState** method sets a single Direct3DDevice lighting-related state value.

```
HRESULT SetLightState(
  D3DLIGHTSTATETYPE dwLightStateType,
  DWORD dwLightState
);
```

Parameters

dwLightStateType
Device state variable that is being modified. This parameter can be any of the members of the **D3DLIGHTSTATETYPE** enumerated type.

dwLightState
New value for the Direct3DDevice light state. The meaning of this parameter is dependent on the value specified for *dwLightStateType*. For example, if *dwLightStateType* were **D3DLIGHTSTATE_COLORMODEL**, the second parameter would be one of the values of the **D3DCOLORMODEL** data type.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::GetLightState, **IDirect3DDevice3::SetRenderState**, **IDirect3DDevice3::SetTransform**

IDirect3DDevice3::SetRenderState

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::SetRenderState** method sets a single Direct3DDevice rendering state parameter.

```
HRESULT SetRenderState(
  D3DRENDERSTATETYPE dwRenderStateType,
  DWORD dwRenderState
);
```

Parameters

dwRenderStateType
Device state variable that is being modified. This parameter can be any of the members of the **D3DRENDERSTATETYPE** enumerated type.

dwRenderState
New value for the Direct3DDevice render state. The meaning of this parameter is dependent on the value specified for *dwRenderStateType*. For example, if *dwRenderStateType* were D3DRENDERSTATE_SHADEMODE, the second parameter would be one of the members of the **D3DSHADEMODE** enumerated type.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

Applications should use the **IDirect3DDevice3::SetTextureStageState** method to set texture states in favor of the legacy texture-related render states. For more information, see About Render States.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::GetRenderState, **IDirect3DDevice3::SetLightState**, **IDirect3DDevice3::SetTransform**

IDirect3DDevice3::SetRenderTarget

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::SetRenderTarget** method permits the application to easily route rendering output to a new DirectDraw surface as a render target.

```
HRESULT SetRenderTarget(
  LPDIRECTDRAWSURFACE4 lpNewRenderTarget,
  DWORD dwFlags
);
```

Parameters

lpNewRenderTarget
Address of a **IDirectDrawSurface4** interface for the surface object that will be the new rendering target. This surface must be created with the DDSCAPS_3DDEVICE capability.

dwFlags
Not currently used; set to zero.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The error may be one of the following values:

DDERR_INVALIDPARAMS	One of the arguments is invalid.
DDERR_INVALIDSURFACETYPE	The surface passed as the first parameter is invalid.

Remarks

You cannot use this method to set a new render target surface with a depth-buffer if the current render target does not have a depth buffer. Likewise, you cannot use this method to switch from a non-depth-buffered render target to a depth-buffered render target. Attempts to do this will fail in debug builds, and can exhibit unreliable behavior in retail builds. Given that both the new and old render targets use depth buffers, the depth-buffer attached to the new render target replaces the previous depth-buffer for the context.

When you change the rendering target, all of the handles associated with the previous rendering target become invalid. This means that you will have to reacquire all of the texture handles. If you are using ramp mode, you should also update the texture handles inside materials, by calling the **IDirect3DMaterial3::SetMaterial** method. Any execute buffers (which have embedded handles) also need to be updated. The **IDirect3DDevice3::SetRenderTarget** method is most useful to applications that use

the DrawPrimitive methods, especially when these applications do not use ramp mode.

If the new render target surface has different dimensions from the old (length, width, pixel-format), this method marks the viewport as invalid. The viewport may be revalidated after calling **IDirect3DDevice3::SetRenderTarget** by calling **IDirect3DViewport3::SetViewport** to restate viewport parameters that are compatible with the new surface.

Capabilities do not change with changes in the properties of the render target surface. Both the Direct3D HAL and the software rasterizers have only one opportunity to expose capabilities to the application. The system cannot expose different sets of capabilities depending on the format of the destination surface.

If more than one depth-buffer is attached to the render target, this function fails.

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::GetRenderTarget

IDirect3DDevice3::SetTexture

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::SetTexture** method assigns a texture to a given stage for a device.

```
HRESULT  SetTexture(
  DWORD dwStage,
  LPDIRECT3DTEXTURE2 lpTexture
);
```

Parameters

dwStage
Stage identifier to which the texture will be set. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value allowed for *dwStage* is 7.

lpTexture

Address of the **IDirect3DTexture2** interface for the texture being set.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method was introduced with the **IDirect3DDevice3** interface.

Software devices do not support assigning a texture to more than one texture stage at a time.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::GetTexture, **IDirect3DDevice3::GetTextureStageState**, **IDirect3DDevice3::SetTextureStageState**, Textures

IDirect3DDevice3::SetTextureStageState

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::SetTextureStageState** method sets the state value for a currently assigned texture.

```
HRESULT SetTextureStageState(
  DWORD dwStage,
  D3DTEXTURESTAGESTATETYPE dwState,
  DWORD dwValue
);
```

Parameters

dwStage

Stage identifier of the texture for which the state value will be set. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value allowed for *dwStage* is 7.

dwState
Texture state to be set. This parameter can be any member of the **D3DTEXTURESTAGESTATETYPE** enumerated type.

dwValue
State value to be set. The meaning of this value is determined by the *dwState* parameter.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method was introduced with the **IDirect3DDevice3** interface.

Applications should use this method to set texture states in favor of the legacy texture-related render states. For more information, see About Render States.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::GetTextureStageState, **IDirect3DDevice3::GetTexture**, **IDirect3DDevice3::SetTexture**, Textures

IDirect3DDevice3::SetTransform

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::SetTransform** method sets a single Direct3DDevice transformation-related state.

HRESULT SetTransform(
D3DTRANSFORMSTATETYPE *dtstTransformStateType*,

LPD3DMATRIX *lpD3DMatrix*
);

Parameters

dtstTransformStateType
Device state variable that is being modified. This parameter can be any of the members of the **D3DTRANSFORMSTATETYPE** enumerated type.

lpD3DMatrix
Address of a **D3DMATRIX** structure that modifies the current transformation.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::GetTransform, **IDirect3DDevice3::SetLightState**, **IDirect3DDevice3::SetRenderState**

IDirect3DDevice3::ValidateDevice

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::ValidateDevice** method reports the device's ability to render the currently set texture blending operations and arguments in a single pass.

HRESULT ValidateDevice(
LPDWORD *lpdwPasses*
);

Parameters

lpdwPasses
Address that will be filled with the number of rendering passes to complete the desired effect through multipass rendering.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
D3DERR_CONFLICTINGTEXTUREFILTER
D3DERR_CONFLICTINGTEXTUREPALETTE
D3DERR_TOOMANYOPERATIONS
D3DERR_UNSUPPORTEDALPHAARG
D3DERR_UNSUPPORTEDALPHAOPERATION
D3DERR_UNSUPPORTEDCOLORARG
D3DERR_UNSUPPORTEDCOLOROPERATION
D3DERR_UNSUPPORTEDFACTORVALUE
D3DERR_UNSUPPORTEDTEXTUREFILTER
D3DERR_WRONGTEXTUREFORMAT

Remarks

Current hardware does not necessarily implement all possible combinations of operations and arguments. You can determine whether a particular blending operation can be performed with given arguments by setting-up the desired blending operation, then calling the **ValidateDevice** method.

The **ValidateDevice** method uses the currently set render states, textures, and, texture stage states to perform validation at the time of the call. Any changes to these factors after the call invalidate the previous result, and the method must be called again before rendering a scene.

Using diffuse iterated values, either as an argument or as an operation (D3DTA_DIFFUSE or **D3DTOP_BLENDDIFFUSEALPHA**) is sparsely supported on current hardware. Most hardware can only introduce iterated color data at the last texture operation stage.

Try to specify the texture (D3DTA_TEXTURE) for each stage as the first argument, in preference to the second argument.

Many cards do not support use of diffuse or scalar values at arbitrary texture stages. Often, these are only available at the first or last texture blending stage.

Many cards do not actually have a blending unit associated with the first texture that is capable of more than replicating alpha to color channels, or inverting the input. As a result, your application might need to use only the second texture stage if possible. On such hardware, the first unit is presumed to be in its default state, which has the first color argument set to D3DTA_TEXTURE with the **D3DTOP_SELECTARG1** operation.

Operations on the output alpha that are more intricate than or substantially different from the color operations are less likely to be supported.

Some hardware does not support simultaneous use of both D3DTA_TFACTOR and D3DTA_DIFFUSE.

Many cards do not support simultaneous use of multiple textures and mipmapped trilinear filtering. If trilinear filtering has been requested for a texture involved in multi-texture blending operations and validation fails, turn off trilinear filtering and revalidate. In this case, it might be best to perform multipass rendering instead.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::GetTextureStageState,
IDirect3DDevice3::SetTextureStageState

IDirect3DDevice3::Vertex

[This is preliminary documentation and subject to change.]

The **IDirect3DDevice3::Vertex** method adds a new Direct3D vertex to the primitive sequence started with a previous call to the **IDirect3DDevice3::Begin** method.

```
HRESULT Vertex(
  LPVOID lpVertex
);
```

Parameters

lpVertex
Pointer to the next Direct3D vertex to be added to the currently started primitive sequence. This can be any of the Direct3D vertex types (**D3DLVERTEX**, **D3DTLVERTEX**, or **D3DVERTEX**) or a vertex specified in flexible vertex

format. The vertex format must match the description specified in the preceding call to **IDirect3DDevice3::Begin**.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DDevice3::Begin, **IDirect3DDevice3::End**

IDirect3DExecuteBuffer

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirect3DExecuteBuffer** interface to set up and control Direct3D execute buffers. This section is a reference to the methods of this interface. For a conceptual overview, see Execute Buffers.

The methods of the **IDirect3DExecuteBuffer** interface can be organized into the following groups:

Execute data	**GetExecuteData**
	SetExecuteData
Lock and unlock	**Lock**
	Unlock

Miscellaneous	**Initialize**
	Optimize
	Validate

The **IDirect3DExecuteBuffer** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DEXECUTEBUFFER** type is defined as a pointer to the **IDirect3DExecuteBuffer** interface:

```
typedef struct IDirect3DExecuteBuffer    *LPDIRECT3DEXECUTEBUFFER;
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

Execute Buffers

IDirect3DExecuteBuffer::GetExecuteData

[This is preliminary documentation and subject to change.]

The **IDirect3DExecuteBuffer::GetExecuteData** method retrieves the execute data state of the Direct3DExecuteBuffer object. The execute data is used to describe the contents of the Direct3DExecuteBuffer object.

```
HRESULT GetExecuteData(
  LPD3DEXECUTEDATA lpData
);
```

Parameters

lpData
Address of a **D3DEXECUTEDATA** structure that will be filled with the current execute data state of the Direct3DExecuteBuffer object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_LOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This call fails if the Direct3DExecuteBuffer object is locked.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DExecuteBuffer::SetExecuteData

IDirect3DExecuteBuffer::Initialize

[This is preliminary documentation and subject to change.]

The **IDirect3DExecuteBuffer::Initialize** method is provided for compliance with the COM protocol.

```
HRESULT Initialize(
  LPDIRECT3DDEVICE lpDirect3DDevice,
  LPD3DEXECUTEBUFFERDESC lpDesc
);
```

Parameters

lpDirect3DDevice
Address of the device representing the Direct3D object.

lpDesc
Address of a **D3DEXECUTEBUFFERDESC** structure that describes the Direct3DExecuteBuffer object to be created. The call fails if a buffer of at least the specified size cannot be created.

Return Values

The method returns DDERR_ALREADYINITIALIZED because the Direct3DExecuteBuffer object is initialized when it is created.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirect3DExecuteBuffer::Lock

[This is preliminary documentation and subject to change.]

The **IDirect3DExecuteBuffer::Lock** method obtains a direct pointer to the commands in the execute buffer.

```
HRESULT Lock(
  LPD3DEXECUTEBUFFERDESC lpDesc
);
```

Parameters

lpDesc
Address of a **D3DEXECUTEBUFFERDESC** structure. When the method returns, the **lpData** member will be set to point to the actual data to which the application has access. This data may reside in system or video memory, and is specified by the **dwCaps** member. The application may use the **IDirect3DExecuteBuffer::Lock** method to request that Direct3D move the data between system or video memory.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_LOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_WASSTILLDRAWING

Remarks

This call fails if the Direct3DExecuteBuffer object is locked—that is, if another thread is accessing the buffer, or if a **IDirect3DDevice::Execute** method that was issued on this buffer has not yet completed.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DExecuteBuffer::Unlock

IDirect3DExecuteBuffer::Optimize

[This is preliminary documentation and subject to change.]

The **IDirect3DExecuteBuffer::Optimize** method is not currently supported.

```
HRESULT Optimize();
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirect3DExecuteBuffer::SetExecuteData

[This is preliminary documentation and subject to change.]

The **IDirect3DExecuteBuffer::SetExecuteData** method sets the execute data state of the Direct3DExecuteBuffer object. The execute data is used to describe the contents of the Direct3DExecuteBuffer object.

```
HRESULT SetExecuteData(
  LPD3DEXECUTEDATA lpData
);
```

Parameters

lpData
Address of a **D3DEXECUTEDATA** structure that describes the execute buffer layout.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_LOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This call fails if the Direct3DExecuteBuffer object is locked.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DExecuteBuffer::GetExecuteData

IDirect3DExecuteBuffer::Unlock

[This is preliminary documentation and subject to change.]

The **IDirect3DExecuteBuffer::Unlock** method releases the direct pointer to the commands in the execute buffer. This must be done prior to calling the **IDirect3DDevice::Execute** method for the buffer.

```
HRESULT Unlock();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_NOT_LOCKED
DDERR_INVALIDOBJECT

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DExecuteBuffer::Lock

IDirect3DExecuteBuffer::Validate

[This is preliminary documentation and subject to change.]

The **IDirect3DExecuteBuffer::Validate** method is not currently implemented.

```
HRESULT Validate(
  LPDWORD lpdwOffset,
  LPD3DVALIDATECALLBACK lpFunc,
  LPVOID lpUserArg,
  DWORD dwReserved
);
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirect3DLight

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirect3DLight** interface to retrieve and set the capabilities of lights. This section is a reference to the methods of this interface. For a conceptual overview, see Lights.

The **IDirect3DLight** interface is obtained by calling the **IDirect3D3::CreateLight** method.

The methods of the **IDirect3DLight** interface can be organized into the following groups:

Get and set	**GetLight**
	SetLight
Initialization	**Initialize**

The **IDirect3DLight** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DLIGHT** type is defined as a pointer to the **IDirect3DLight** interface:

```
typedef struct IDirect3DLight    *LPDIRECT3DLIGHT;
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirect3DLight::GetLight

[This is preliminary documentation and subject to change.]

The **IDirect3DLight::GetLight** method retrieves the light information for the Direct3DLight object.

```
HRESULT GetLight(
  LPD3DLIGHT lpLight
);
```

Parameters

lpLight
Address of a **D3DLIGHT2** structure that will be filled with the current light data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

Although this method's declaration specifies the *lpLight* parameter as being the address of a **D3DLIGHT** structure, that structure is not normally used. Rather, the **D3DLIGHT2** structure is recommended to achieve the best lighting effects.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DLight::SetLight

IDirect3DLight::Initialize

[This is preliminary documentation and subject to change.]

The **IDirect3DLight::Initialize** method is provided for compliance with the COM protocol.

```
HRESULT Initialize(
  LPDIRECT3D lpDirect3D
);
```

Parameters

lpDirect3D
Address of the **IDirect3D** interface for the Direct3D object.

Return Values

The method returns DDERR_ALREADYINITIALIZED because the Direct3DLight object is initialized when it is created.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirect3DLight::SetLight

[This is preliminary documentation and subject to change.]

The **IDirect3DLight::SetLight** method sets the light information for the Direct3DLight object.

```
HRESULT SetLight(
  LPD3DLIGHT lpLight
);
```

Parameters

lpLight
Address of a **D3DLIGHT2** structure that will be used to set the current light data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

Although this method's declaration specifies the *lpLight* parameter as being the address of a **D3DLIGHT** structure, that structure is not normally used. Rather, the **D3DLIGHT2** structure is recommended to achieve the best lighting effects.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DLight::GetLight

IDirect3DMaterial3

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirect3DMaterial3** interface to retrieve and set the properties of materials. This section is a reference to the methods of this interface. For a conceptual overview, see Materials.

You create this interface by calling the **IDirect3D3::CreateMaterial** method. The **IDirect3DMaterial3** interface provides identical services to its predecessor, the **IDirect3DMaterial2** interface, but binds materials to the latest iteration of the Direct3DDevice interface, **IDirect3DDevice3**.

The methods of the **IDirect3DMaterial3** interface can be organized into the following groups:

Handles	**GetHandle**
Materials	**GetMaterial**
	SetMaterial

The **IDirect3DMaterial3** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DMATERIAL3**, **LPDIRECT3DMATERIAL2**, and **LPDIRECT3DMATERIAL** types are defined as pointers to the **IDirect3DMaterial3**, **IDirect3DMaterial2** and **IDirect3DMaterial** interfaces:

```
typedef struct IDirect3DMaterial3    *LPDIRECT3DMATERIAL3;
typedef struct IDirect3DMaterial2    *LPDIRECT3DMATERIAL2;
typedef struct IDirect3DMaterial     *LPDIRECT3DMATERIAL;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Materials, Lighting and Materials

IDirect3DMaterial3::GetHandle

[This is preliminary documentation and subject to change.]

The **IDirect3DMaterial3::GetHandle** method binds a material to a device, retrieving a handle that represents the association between the two. This handle is used in all Direct3D methods in which a material is to be referenced. A material can be used by only one device at a time.

If the device is destroyed, the material is disassociated from the device.

```
HRESULT GetHandle(
  LPDIRECT3DDEVICE3 lpDirect3DDevice,
  LPD3DMATERIALHANDLE lpHandle
);
```

Parameters

lpDirect3DDevice
Address of the **IDirect3DDevice3** interface for the rendering device to which the material is being bound.

lpHandle
Address of a variable that will be filled with the material handle corresponding to the Direct3DMaterial object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is DDERR_INVALIDOBJECT.

Remarks

In the **IDirect3DMaterial2** interface, this method accepts a pointer to an **IDirect3DDevice2** interface instead of an **IDirect3DDevice3** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Retrieving Material Handles

IDirect3DMaterial3::GetMaterial

[This is preliminary documentation and subject to change.]

The **IDirect3DMaterial3::GetMaterial** method retrieves the material data for the Direct3DMaterial object.

```
HRESULT GetMaterial(
  LPD3DMATERIAL lpMat
);
```

Parameters

lpMat
Address of a **D3DMATERIAL** structure that will be filled with the current material properties.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DMaterial2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DMaterial3::SetMaterial, Retrieving Material Properties, Setting Material Properties

IDirect3DMaterial3::SetMaterial

[This is preliminary documentation and subject to change.]

The **IDirect3DMaterial3::SetMaterial** method sets the material data for a material object.

```
HRESULT SetMaterial(
  LPD3DMATERIAL lpMat
);
```

Parameters

lpMat
Address of a **D3DMATERIAL** structure that contains the material properties.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DMaterial3::GetMaterial, Retrieving Material Properties, Setting Material Properties

IDirect3DTexture2

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirect3DTexture2** interface to retrieve and set the properties of textures. This section is a reference to the methods of this interface. For a conceptual overview, see Textures.

You create the **IDirect3DTexture2** interface by calling the **IDirectDrawSurface::QueryInterface** method from the DirectDrawSurface object that was created as a texture map.

The methods of the **IDirect3DTexture2** interface can be organized into the following groups:

Handles	**GetHandle**
Loading	**Load**
Palette information	**PaletteChanged**

The **IDirect3DTexture2** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DTEXTURE2** and **LPDIRECT3DTEXTURE** types are defined as pointers to the **IDirect3DTexture2** and **IDirect3DTexture** interfaces:

```
typedef struct IDirect3DTexture2    *LPDIRECT3DTEXTURE2;
typedef struct IDirect3DTexture     *LPDIRECT3DTEXTURE;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Textures

IDirect3DTexture2::GetHandle

[This is preliminary documentation and subject to change.]

The **IDirect3DTexture2::GetHandle** method obtains the texture handle to be used when rendering with the **IDirect3DDevice2** or **IDirect3DDevice** interfaces.

```
HRESULT GetHandle(
  LPDIRECT3DDEVICE2 lpDirect3DDevice2,
  LPD3DTEXTUREHANDLE lpHandle
);
```

Parameters

lpDirect3DDevice2
Address of the Direct3DDevice2 object into which the texture is to be loaded.

lpHandle
Address that will contain the texture handle corresponding to the Direct3DTexture2 object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDPARAMS

Remarks

In the **IDirect3DTexture** interface, this method uses a pointer to a Direct3DDevice object instead of a Direct3DDevice2 object.

Texture handles are used only device interfaces earlier than **IDirect3DDevice3**. The **IDirect3DDevice3** interface references textures using texture interface pointers, set through the **IDirect3DDevice3::SetTexture** method.

You cannot use this method to retrieve the handle of a texture that is managed by Direct3D. For more information, see Automatic Texture Management .

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DTexture2::Load

[This is preliminary documentation and subject to change.]

The **IDirect3DTexture2::Load** method loads a system-memory texture surface into a video-memory texture surface. This method can be used to load texture mipmap chains (see remarks).

```
HRESULT Load(
  LPDIRECT3DTEXTURE2 lpD3DTexture2
);
```

Parameters

lpD3DTexture2
Address of the texture to load.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return values, see Direct3D Immediate Mode Return Values.

Remarks

This method uses hardware-accelerated blit operations to load data from the source texture into the destination texture.

If both textures are mipmaps, the method will copy the mipmap levels from the source mipmap that match those of the destination mipmap. If the destination mipmap uses levels-of-detail not present in the source mipmap, the method fails.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DTexture2::PaletteChanged

[This is preliminary documentation and subject to change.]

The **IDirect3DTexture2::PaletteChanged** method informs the driver that the palette has changed on a texture surface.

```
HRESULT PaletteChanged(
```

```
  DWORD dwStart,
  DWORD dwCount
);
```

Parameters

dwStart
Index of first palette entry that has changed.

dwCount
Total number of palette entries that have changed.

Return Values

This method returns D3D_OK.

If the method fails, the return value is an error. For a list of possible return values, see Direct3D Immediate Mode Return Values.

Remarks

This method is particularly useful for applications that play video clips and therefore require palette-changing capabilities.

This method only affects the legacy ramp device. For all other devices, this method takes no action and returns D3D_OK.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DVertexBuffer

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirect3DVertexBuffer** interface to manipulate a collection of vertices for use with the **IDirect3DDevice3::DrawPrimitiveVB** and **IDirect3DDevice3::DrawIndexedPrimitiveVB** rendering methods. This section is a reference to the methods of this interface. For a conceptual overview, see Vertex Buffers.

This methods of the **IDirect3DVertexBuffer** interface can be organized into the following groups:

Information	**GetVertexBufferDesc**

Vertex data	**Lock**
	Optimize
	ProcessVertices
	Unlock

The **IDirect3DVertexBuffer** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DVERTEXBUFFER** data type is defined as a pointer to the **IDirect3DVertexBuffer** interface:

```
typedef struct IDirect3DVertexBuffer *LPDIRECT3DVERTEXBUFFER
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Vertex Buffers

IDirect3DVertexBuffer::GetVertexBufferDesc

[This is preliminary documentation and subject to change.]

The **IDirect3DVertexBuffer::GetVertexBufferDesc** method retrieves a description of the vertex buffer.

```
HRESULT GetVertexBufferDesc(
  LPD3DVERTEXBUFFERDESC lpVBDesc,
);
```

Parameters

lpVBDesc

Address of a **D3DVERTEXBUFFERDESC** structure that will be filled with a description of the vertex buffer.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be DDERR_INVALIDPARAMS or another error code.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DVertexBuffer::Lock

[This is preliminary documentation and subject to change.]

The **IDirect3DVertexBuffer::Lock** methods locks a vertex buffer and obtains a pointer to the vertex buffer memory.

```
HRESULT Lock(
  DWORD   dwFlags,
  LPVOID* lplpData,
  LPDWORD lpdwSize
);
```

Parameters

dwFlags
Flags indicating how the vertex buffer memory should be locked.

DDLOCK_EVENT
This flag is not currently implemented.

DDLOCK_NOSYSLOCK
If possible, do not take the Win16Mutex (also known as Win16Lock).

DDLOCK_READONLY
Indicates that the memory being locked will only be read from.

DDLOCK_SURFACEMEMORYPTR
Indicates that a valid memory pointer to the vertex buffer should be returned; this is the default.

DDLOCK_WAIT
If a lock cannot be obtained immediately, the method retries until a lock is obtained or another error occurs.

DDLOCK_WRITEONLY
Indicates that the memory being locked will only be written to.

lplpData
Address of a variable that will contain the address of the vertex buffer memory if the call succeeds.

lpdwSize
Address of a variable that will contain the size of the vertex buffer memory at *lplpData*. Set to NULL if the buffer size is unneeded.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_VERTEXBUFFEROPTIMIZED
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACEBUSY
DDERR_SURFACELOST

Remarks

After locking the vertex buffer, you can access the memory until a corresponding call to **IDirect3DVertexBuffer::Unlock**.

You cannot render from a locked vertex buffer; calls to the **IDirect3DDevice3::DrawIndexedPrimitiveVB** or **IDirect3DDevice3::DrawPrimitiveVB** method using a locked buffer will fail, returning D3DERR_VERTEXBUFFERLOCKED.

This method often causes the system to hold the Win16Mutex until you call the **IDirect3DVertexBuffer::Unlock** method. GUI debuggers cannot operate while the Win16Mutex is held.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DVertexBuffer::Unlock, Accessing Vertex Buffer Memory

IDirect3DVertexBuffer::Optimize

[This is preliminary documentation and subject to change.]

The **IDirect3DVertexBuffer::Optimize** method converts an unoptimized vertex buffer into an optimized vertex buffer.

```
HRESULT Optimize (
  LPDIRECT3DDEVICE3 lpD3DDevice,
  DWORD dwFlags
);
```

Parameters

lpD3DDevice
Address of the **IDirect3DDevice3** interface of the device for which this vertex buffer will be optimized.

dwFlags
Not currently used; set to zero.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_VERTEXBUFFEROPTIMIZED
D3DERR_VERTEXBUFFERLOCKED
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Optimizing a Vertex Buffer, Vertex Buffers

IDirect3DVertexBuffer::ProcessVertices

[This is preliminary documentation and subject to change.]

The **IDirect3DVertexBuffer::ProcessVertices** method processes untransformed vertices into a transformed or optimized vertex buffer.

```
HRESULT ProcessVertices(
  DWORD dwVertexOp,
  DWORD dwDestIndex,
  DWORD dwCount,
  LPDIRECT3DVERTEXBUFFER lpSrcBuffer,
  DWORD dwSrcIndex,
  LPDIRECT3DDEVICE3      lpD3DDevice,
  DWORD dwFlags
);
```

Parameters

dwVertexOp

Flags defining how the method processes the vertices as they are transferred from the source buffer. You can specify any combination of the following flags:

D3DVOP_CLIP

Transform the vertices and clip any vertices that exist outside the viewing frustum. This flag cannot be used with vertex buffers that do not contain clipping information (for example, created with the D3DDP_DONOTCLIP flag).

D3DVOP_EXTENTS

Transform the vertices, then update the extents of the screen rectangle when the vertices are rendered. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice3::GetClipStatus** will not have been updated to account for the vertices when they are rendered.

D3DVOP_LIGHT

Light the vertices.

D3DVOP_TRANSFORM

Transform the vertices using the world, view, and projection matrices. This flag must always be set.

dwDestIndex

Index into the destination vertex buffer (this buffer) where the vertices will be placed after processing.

dwCount

Number of vertices in the source buffer to process.

lpSrcBuffer

Address of the **IDirect3DVertexBuffer** interface for the source vertex buffer.

dwSrcIndex

Index of the first vertex in the source buffer to be processed.

lpD3DDevice

Address of the **IDirect3DDevice3** interface for the device that will be used to transform the vertices.

dwFlags

Reserved for future use; set to zero. Failing to set this parameter to 0 causes the method to fail and return DDERR_INVALIDPARAMS.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDVERTEXFORMAT
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACEBUSY
DDERR_SURFACELOST

Remarks

You must always include the D3DVOP_TRANSFORMED flag in the *dwVertexOp* parameter. If you fail to include this flag, the method will fail, returning DDERR_INVALIDPARAMS.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Processing Vertices, Vertex Buffers

IDirect3DVertexBuffer::Unlock

[This is preliminary documentation and subject to change.]

The **IDirect3DVertexBuffer::Unlock** method unlocks a previously locked vertex buffer.

```
HRESULT Unlock();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACEBUSY
DDERR_SURFACELOST

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DVertexBuffer::Lock, Accessing Vertex Buffer Memory

IDirect3DViewport3

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirect3DViewport3** interface to retrieve and set the properties of viewports. This section is a reference to the methods of this interface. For a conceptual overview, see Viewports and Clipping.

The **IDirect3DViewport3** interface offers the same services as the **IDirect3DViewport2** interface, but adds the **Clear2** method, which simultaneously clears the viewport, depth-buffer, and stencil buffer.

You create the **IDirect3DViewport3** interface by calling the **IDirect3D3::CreateViewport** method.

The methods of the **IDirect3DViewport3** interface can be organized into the following groups:

Backgrounds	**GetBackground**
	GetBackgroundDepth
	GetBackgroundDepth2
	SetBackground
	SetBackgroundDepth
	SetBackgroundDepth2

Lights	**AddLight**
	DeleteLight
	LightElements
	NextLight
Materials and viewports	**Clear**
	Clear2
	GetViewport
	GetViewport2
	SetViewport
	SetViewport2
Miscellaneous	**Initialize**
Transformation	**TransformVertices**

The **IDirect3DViewport3** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **LPDIRECT3DVIEWPORT3, LPDIRECT3DVIEWPORT2** and **LPDIRECT3DVIEWPORT** types are defined as pointers to the **IDirect3DViewport3, IDirect3DViewport2,** and **IDirect3DViewport** interfaces:

```
typedef struct IDirect3DViewport3    *LPDIRECT3DVIEWPORT3;
typedef struct IDirect3DViewport2    *LPDIRECT3DVIEWPORT2;
typedef struct IDirect3DViewport     *LPDIRECT3DVIEWPORT;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

Viewports and Clipping

IDirect3DViewport3::AddLight

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::AddLight** method adds the specified light to the list of Direct3DLight objects associated with this viewport and increments the reference count of the light object.

```
HRESULT AddLight(
  LPDIRECT3DLIGHT lpDirect3DLight
);
```

Parameters

lpDirect3DLight
Address of the **IDirect3DLight** interface for the light that should be associated with this viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DViewport3::Clear

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::Clear** method clears the viewport or a set of rectangles in the viewport to the current background material.

```
HRESULT Clear(
```

```
  DWORD     dwCount,
  LPD3DRECT lpRects,
  DWORD     dwFlags
);
```

Parameters

dwCount
Number of rectangles pointed to by *lpRects*.

lpRects
Address of an array of **D3DRECT** structures. Each rectangle uses screen coordinates that correspond to points on the render target surface. Coordinates are clipped to the bounds of the viewport rectangle.

dwFlags
Flags indicating what to clear: the rendering target, the depth-buffer, or both.

D3DCLEAR_TARGET
Clear the rendering target to the background material (if set).

D3DCLEAR_ZBUFFER
Clear the depth-buffer or set it to the current background depth field (if set).

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_VIEWPORTHASNODEVICE

D3DERR_ZBUFFER_NOTPRESENT

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

The behavior of this method is undefined for depth buffers that include stencil bits, and using this method on such a depth buffer can cause the stencil bits to be arbitrarily overwritten or the depth values to be incorrect. Always use the use the **IDirect3DViewport3::Clear2** method to clear depth buffers that contain stencil bits.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::Clear2

IDirect3DViewport3::Clear2

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::Clear2** method clears the viewport (or a set of rectangles in the viewport) to a specified RGBA color, clears the depth-buffer, and erases the stencil buffer.

```
HRESULT Clear2(
  DWORD     dwCount,
  LPD3DRECT lpRects,
  DWORD     dwFlags,
  DWORD     dwColor,
  D3DVALUE  dvZ,
  DWORD     dwStencil
);
```

Parameters

dwCount
Number of rectangles in the array at *lpRects*.

lpRects
Array of **D3DRECT** structures that describe the rectangles to be cleared. Set a rectangle to the dimensions of the rendering target to clear the entire surface. Each rectangle uses screen coordinates that correspond to points on the render target surface. Coordinates are clipped to the bounds of the viewport rectangle.

dwFlags
Flags indicating which surfaces should be cleared. This parameter can be any combination of the following flags, but at least one flag must be used:

D3DCLEAR_TARGET
Clear the rendering target to the color in the *dwColor* parameter.

D3DCLEAR_STENCIL
Clear the stencil buffer to the value in the *dwStencil* parameter.

D3DCLEAR_ZBUFFER
Clear the depth-buffer to the value in the *dvZ* parameter.

dwColor
32-bit RGBA color value to which the render target surface will be cleared.

dvZ

New z value that this method stores in the depth-buffer. This parameter can range from 0.0 to 1.0, inclusive. The value of 0.0 represents the nearest distance to the viewer, and 1.0 the farthest distance.

dwStencil

Integer value to store in each stencil buffer entry. This parameter can range from 0 to 2^n-1 inclusive, where *n* is the bit depth of the stencil buffer.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_STENCILBUFFER_NOTPRESENT
D3DERR_VIEWPORTHASNODEVICE
D3DERR_ZBUFFER_NOTPRESENT
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method fails if you specify the D3DCLEAR_ZBUFFER or D3DCLEAR_STENCIL flags when the render target does not have an attached depth-buffer. This behavior differs from the **IDirect3DViewport3::Clear** method, which will succeed if under these circumstances.

If you specify the D3DCLEAR_STENCIL flag when the depth-buffer format doesn't contain stencil buffer information, this method fails.

This method ignores the current background material for the viewport; to clear a viewport using the background material, use the **IDirect3DViewport3::Clear** method.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::Clear

IDirect3DViewport3::DeleteLight

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::DeleteLight** method removes the specified light from the list of Direct3DLight objects associated with this viewport, and decrements the reference count for the light object.

```
HRESULT DeleteLight(
  LPDIRECT3DLIGHT lpDirect3DLight
);
```

Parameters

lpDirect3DLight
Address of the **IDirect3DLight** interface for the light that should be disassociated with this viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DViewport3::GetBackground

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::GetBackground** method retrieves the handle to a material that represents the current background associated with the viewport.

```
HRESULT GetBackground(
  LPD3DMATERIALHANDLE lphMat,
  LPBOOL lpValid
);
```

Parameters

lphMat
Address that will contain the handle to the material being used as the background.

lpValid
Address of a variable that will be filled to indicate whether a background is associated with the viewport. If this parameter is FALSE, no background is associated with the viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::SetBackground

IDirect3DViewport3::GetBackgroundDepth

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::GetBackgroundDepth** method retrieves a DirectDraw surface that represents the current background-depth field associated with the viewport.

HRESULT GetBackgroundDepth(
LPDIRECTDRAWSURFACE* *lplpDDSurface*,

LPBOOL *lpValid*
);

Parameters

lplpDDSurface
Address of a variable that will be filled with the **IDirectDrawSurface** interface for the surface object that represents the background depth.

lpValid
Address of a variable that is set to FALSE if no background depth is associated with the viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::SetBackgroundDepth

IDirect3DViewport3::GetBackgroundDepth2

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::GetBackgroundDepth2** method retrieves a DirectDraw surface that represents the current background-depth field associated with the viewport.

```
HRESULT GetBackgroundDepth2(
  LPDIRECTDRAWSURFACE4* lplpDDS,
  LPBOOL lpValid
);
```

Parameters

lplpDDS
Address of a variable that will be filled with the **IDirectDrawSurface4** interface for the surface object that represents the background depth.

lpValid
Address of a variable that is set to FALSE if no background depth is associated with the viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::SetBackgroundDepth2

IDirect3DViewport3::GetViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::GetViewport** method retrieves the viewport registers of the viewport. This method is provided for backward compatibility. It has been superseded by the **IDirect3DViewport3::GetViewport2** method.

```
HRESULT GetViewport(
  LPD3DVIEWPORT lpData
);
```

Parameters

lpData
Address of a **D3DVIEWPORT** structure representing the viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::GetViewport2, **IDirect3DViewport3::SetViewport**

IDirect3DViewport3::GetViewport2

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::GetViewport2** method retrieves the viewport registers of the viewport.

```
HRESULT GetViewport2(
  LPD3DVIEWPORT2 lpData
);
```

Parameters

lpData
Address of a **D3DVIEWPORT2** structure representing the viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::SetViewport2

IDirect3DViewport3::Initialize

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::Initialize** method is not implemented.

```
HRESULT Initialize(
  LPDIRECT3D lpDirect3D
);
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DViewport3::LightElements

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::LightElements** method is not currently implemented.

```
HRESULT LightElements(
  DWORD dwElementCount,
  LPD3DLIGHTDATA lpData
);
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DViewport3::NextLight

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::NextLight** method enumerates the Direct3DLight objects associated with the viewport.

```
HRESULT NextLight(
  LPDIRECT3DLIGHT lpDirect3DLight,
  LPDIRECT3DLIGHT* lplpDirect3DLight,
  DWORD dwFlags
);
```

Parameters

lpDirect3DLight
Address of a light in the list of lights associated with this viewport object.

lplpDirect3DLight
Address of a pointer that will contain the requested light in the list of lights associated with this viewport object. The requested light is specified in the *dwFlags* parameter.

dwFlags
Flags specifying which light to retrieve from the list of lights. This must be set to one of the following flags:

D3DNEXT_HEAD
Retrieve the item at the beginning of the list.

D3DNEXT_NEXT
Retrieve the next item in the list.

D3DNEXT_TAIL
Retrieve the item at the end of the list.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

IDirect3DViewport3::SetBackground

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::SetBackground** method sets the background material associated with the viewport.

```
HRESULT SetBackground(
  D3DMATERIALHANDLE hMat
);
```

Parameters

hMat
Material handle that will be used as the background.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::GetBackground

IDirect3DViewport3::SetBackgroundDepth

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::SetBackgroundDepth** method sets the background-depth field for the viewport.

```
HRESULT SetBackgroundDepth(
  LPDIRECTDRAWSURFACE lpDDSurface
);
```

Parameters

lpDDSurface
Address of the DirectDrawSurface object representing the background depth.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

The depth-buffer is filled with the specified depth field when the **IDirect3DViewport3::Clear** method is called and the D3DCLEAR_ZBUFFER flag is specified. The bit depth must be 16 bits.

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::GetBackgroundDepth

IDirect3DViewport3::SetBackgroundDepth2

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::SetBackgroundDepth2** method sets the background-depth field for the viewport.

```
HRESULT SetBackgroundDepth2(
  LPDIRECTDRAWSURFACE4 lpDDS
);
```

Parameters

lpDDS
Address of the **IDirectDrawSurface4** interface for the surface object that represents the new background depth.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

The depth-buffer is filled with the specified depth field when the **IDirect3DViewport3::Clear** or **IDirect3DViewport3::Clear2** methods are called with the D3DCLEAR_ZBUFFER flag is specified.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::GetBackgroundDepth2

IDirect3DViewport3::SetViewport

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::SetViewport** method sets the viewport registers of the viewport. This method is provided for backward compatibility. It has been superseded by the **IDirect3DViewport3::SetViewport2** method.

```
HRESULT SetViewport(
  LPD3DVIEWPORT lpData
);
```

Parameters

lpData
Address of a **D3DVIEWPORT** structure that describces the new viewport properties. The method ignores the values in the **dvMaxX**, **dvMaxY**, **dvMinZ**, and **dvMaxZ** members.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

You cannot set viewport parameters unless the viewport is associated with a rendering device (by calling the **IDirect3DDevice3::AddViewport** method). For details, see Preparing to Use a Viewport.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::GetViewport, **IDirect3DViewport3::SetViewport2**, Using Viewports, Viewports and Clipping

IDirect3DViewport3::SetViewport2

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::SetViewport2** method sets the viewport registers of the viewport.

```
HRESULT SetViewport2(
  LPD3DVIEWPORT2 lpData
);
```

Parameters

lpData
Address of a **D3DVIEWPORT2** structure that contains the new viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_VIEWPORTHASNODEVICE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

You cannot set viewport parameters unless the viewport is associated with a rendering device (by calling the **IDirect3DDevice3::AddViewport** method). For details, see Preparing to Use a Viewport.

The **dvMinZ** and **dvMaxZ** members of the associated **D3DVIEWPORT2** structure must not contain identical values.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

See Also

IDirect3DViewport3::GetViewport2, Using Viewports, Viewports and Clipping

IDirect3DViewport3::TransformVertices

[This is preliminary documentation and subject to change.]

The **IDirect3DViewport3::TransformVertices** method transforms a set of vertices by the transformation matrix.

```
HRESULT TransformVertices(
  DWORD dwVertexCount,
  LPD3DTRANSFORMDATA lpData,
  DWORD dwFlags,
  LPDWORD lpOffscreen
);
```

Parameters

dwVertexCount
Number of vertices in the *lpData* parameter to be transformed.

lpData
Address of a **D3DTRANSFORMDATA** structure that contains the vertices to be transformed. See Remarks.

dwFlags
One of the following flags. See the comments section following the parameter description for a discussion of how to use these flags.

D3DTRANSFORM_CLIPPED
D3DTRANSFORM_UNCLIPPED

lpOffscreen
Address of a variable that is set to a nonzero value if the resulting vertices are all off-screen.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method has often be used to perform visibility checking, as it returns clipping information in the **D3DTRANSFORMDATA** structure after the call completes. Although the clipping information is accurate, visibility checking is best performed by calling the **IDirect3DDevice3::ComputeSphereVisibility** method, which was specially designed and optimized for this purpose.

If the *dwFlags* parameter is set to D3DTRANSFORM_CLIPPED, this method uses the current transformation matrix to transform a set of vertices, checking the resulting vertices to see if they are within the viewing frustum. The homogeneous part of the **D3DTLVERTEX** structure within *lpData* will be set if the vertex is clipped; otherwise only the screen coordinates will be set. The clip intersection of all the vertices transformed is returned in *lpOffscreen*. That is, if *lpOffscreen* is nonzero, all the vertices were off-screen and not straddling the viewport.

Initialize the **drExtent** member of the **D3DTRANSFORMDATA** structure to a **D3DRECT** structure that describes a 2-D bounding rectangle (extents) that the method will "grow" if the transformed vertices do not fit within it. If the transformed vertices are outside the provided extents, the method adjusts the extents to fit the vertices, otherwise no changes are made. If the *dwFlags* parameter is set to D3DTRANSFORM_UNCLIPPED, this method transforms the vertices assuming that the resulting coordinates will be within the viewing frustum. If clipping is requested by setting the *dwFlags* parameter to D3DTRANSFORM_CLIPPED, the method adjusts the extents to fit only the transformed vertices that are within the viewing area.

The **dwClip** member of **D3DTRANSFORMDATA** can help the transformation module determine whether the geometry will need clipping against the viewing volume. Before transforming a geometry, high-level software often can test whether bounding boxes or bounding spheres are wholly within the viewing volume, allowing clipping tests to be skipped, or wholly outside the viewing volume, allowing the geometry to be skipped entirely.

This method is unchanged from its implementation in the **IDirect3DViewport2** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.
Import Library: Use ddraw.lib.

Unimplemented Methods

[This is preliminary documentation and subject to change.]

The following methods were stubs in previous versions of DirectX, but are not implemented in the most recent versions of their interfaces.

IDirect3D::Initialize

IDirect3DMaterial::Initialize

IDirect3DMaterial::Reserve

IDirect3DMaterial::Unreserve

IDirect3DTexture::Initialize

IDirect3DTexture::Unload

D3D_OVERLOADS

[This is preliminary documentation and subject to change.]

C++ programmers who define D3D_OVERLOADS can use the extensions documented here to simplify their code in Direct3D Immediate Mode applications. The use of D3D_OVERLOADS was introduced with DirectX® 5.0. This section is a reference to the D3D_OVERLOADS extensions.

These extensions must be defined with C++ linkage. If D3D_OVERLOADS is defined and the inclusion of D3dtypes.h or D3d.h is surrounded by extern "C", link errors will result. For example, the following syntax would generate link errors because of C linkage of D3D_OVERLOADS functionality:

```
#define D3D_OVERLOADS
extern "C" {
#include <d3d.h>
};
```

The D3D_OVERLOADS extensions can be organized into the following groups:

Constructors	D3DLVERTEX
	D3DTLVERTEX
	D3DVECTOR
	D3DVERTEX
Operators	Access Grant Operators
	Addition Operator
	Assignment Operators
	Bitwise Equality Operator

D3DMATRIX
Division Operator
Multiplication Operator
Subtraction Operator
Unary Operators
Vector Dominance Operators

Helper functions **CrossProduct**
DotProduct
Magnitude
Max
Maximize
Min
Minimize
Normalize
SquareMagnitude

D3D_OVERLOADS Constructors

[This is preliminary documentation and subject to change.]

This section contains reference information for the constructors provided by the D3D_OVERLOADS C++ extensions.

- D3DLVERTEX
- D3DTLVERTEX
- D3DVECTOR
- D3DVERTEX

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

D3DLVERTEX Constructors

[This is preliminary documentation and subject to change.]

The D3D_OVERLOADS constructors for the **D3DLVERTEX** structure offer a convenient way for C++ programmers to create lit vertices.

```
_D3DLVERTEX() { }
_D3DLVERTEX(const D3DVECTOR& v,
            D3DCOLOR _color, D3DCOLOR _specular,
            float _tu, float _tv)
  { x = v.x; y = v.y; z = v.z; dwReserved = 0;
    color = _color; specular = _specular;
    tu = _tu; tv = _tv;
  }
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

D3DTLVERTEX Constructors

[This is preliminary documentation and subject to change.]

The D3D_OVERLOADS constructors for the **D3DTLVERTEX** structure offer a convenient way for C++ programmers to create transformed and lit vertices.

```
_D3DTLVERTEX() { }
_D3DTLVERTEX(const D3DVECTOR& v, float _rhw,
             D3DCOLOR _color, D3DCOLOR _specular,
             float _tu, float _tv)
  { sx = v.x; sy = v.y; sz = v.z; rhw = _rhw;
    color = _color; specular = _specular;
    tu = _tu; tv = _tv;
  }
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

D3DVECTOR Constructors

[This is preliminary documentation and subject to change.]

The D3D_OVERLOADS constructors for the **D3DVECTOR** structure offer a convenient way for C++ programmers to create vectors.

```
_D3DVECTOR() { }
```

```
_D3DVECTOR(D3DVALUE f);
_D3DVECTOR(D3DVALUE _x, D3DVALUE _y, D3DVALUE _z);
_D3DVECTOR(const D3DVALUE f[3]);
```

These constructors are defined as follows:

```
inline _D3DVECTOR::_D3DVECTOR(D3DVALUE f)
   {   x = y = z = f; }

inline _D3DVECTOR::_D3DVECTOR(D3DVALUE _x, D3DVALUE _y, D3DVALUE _z)
   {    x = _x; y = _y; z = _z; }

inline _D3DVECTOR::_D3DVECTOR(const D3DVALUE f[3])
   {    x = f[0]; y = f[1]; z = f[2]; }
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

D3DVERTEX Constructors

[This is preliminary documentation and subject to change.]

The D3D_OVERLOADS constructors for the **D3DVERTEX** structure offer a convenient way for C++ programmers to create lit vertices.

```
_D3DVERTEX() { }
_D3DVERTEX(const D3DVECTOR& v, const D3DVECTOR& n, float _tu, float _tv)
   { x = v.x; y = v.y; z = v.z;
     nx = n.x; ny = n.y; nz = n.z;
     tu = _tu; tv = _tv;
   }
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

D3D_OVERLOADS Operators

[This is preliminary documentation and subject to change.]

This section contains reference information for the operators provided by the D3D_OVERLOADS C++ extensions.

- Access Grant Operators
- Addition Operator
- Assignment Operators
- Bitwise Equality Operator
- D3DMATRIX
- Division Operator
- Multiplication Operator
- Subtraction Operator
- Unary Operators
- Vector Dominance Operators

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

Access Grant Operators (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

The bracket ("[ ]") operators are overloaded operators for the D3D_OVERLOADS extensions. You can use empty brackets ("[ ]") for access grants, "v[0]" to access the x component of a vector, "v[1]" to access the y component, and "v[2]" to access the z component. These operators are defined as follows:

```
const D3DVALUE&operator[ ](int i) const;
D3DVALUE&operator[ ](int i);

inline const D3DVALUE&
_D3DVECTOR::operator[ ](int i) const
{
    return (&x)[i];
}

inline D3DVALUE&
_D3DVECTOR::operator[ ](int i)
{
    return (&x)[i];
```

```
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

Addition Operator (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. The addition operator is defined as follows:

```
_D3DVECTOR operator + (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline _D3DVECTOR
operator + (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return _D3DVECTOR(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

Assignment Operators (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

The assignment operators are overloaded operators for the D3D_OVERLOADS extensions. Both scalar and vector forms of the "*=" and "/=" operators have been implemented. (In the vector form, multiplication and division are memberwise.)

```
_D3DVECTOR& operator += (const _D3DVECTOR& v);
_D3DVECTOR& operator -= (const _D3DVECTOR& v);
_D3DVECTOR& operator *= (const _D3DVECTOR& v);
_D3DVECTOR& operator /= (const _D3DVECTOR& v);
_D3DVECTOR& operator *= (D3DVALUE s);
_D3DVECTOR& operator /= (D3DVALUE s);
```

The assignment operators are defined as follows:

```
inline _D3DVECTOR&
_D3DVECTOR::operator += (const _D3DVECTOR& v)
{
  x += v.x;   y += v.y;   z += v.z;
  return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator -= (const _D3DVECTOR& v)
{
  x -= v.x;   y -= v.y;   z -= v.z;
  return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator *= (const _D3DVECTOR& v)
{
  x *= v.x;   y *= v.y;   z *= v.z;
  return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator /= (const _D3DVECTOR& v)
{
  x /= v.x;   y /= v.y;   z /= v.z;
  return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator *= (D3DVALUE s)
{
  x *= s;   y *= s;   z *= s;
  return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator /= (D3DVALUE s)
{
  x /= s;   y /= s;   z /= s;
  return *this;
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

Bitwise Equality Operator (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. The bitwise-equality operator is defined as follows:

```
int operator == (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline int
operator == (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return v1.x==v2.x && v1.y==v2.y && v1.z == v2.z;
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

D3DMATRIX (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

The D3D_OVERLOADS implementation of the **D3DMATRIX** structure implements a parentheses ("()") operator. This operator offers convenient access to values in the matrix for C++ programmers. Instead of having to refer to the structure members by name, C++ programmers can refer to them by row and column number, and simply index these numbers as needed.

```
typedef struct _D3DMATRIX {
#if (defined __cplusplus) && (defined D3D_OVERLOADS)
    union {
        struct {
#endif

            D3DVALUE        _11, _12, _13, _14;
```

```
        D3DVALUE    _21, _22, _23, _24;
        D3DVALUE    _31, _32, _33, _34;
        D3DVALUE    _41, _42, _43, _44;

#if (defined __cplusplus) && (defined D3D_OVERLOADS)
    };
    D3DVALUE m[4][4];
  };
  _D3DMATRIX() { }

  D3DVALUE& operator()(int iRow, int iColumn) { return m[iRow][iColumn]; }
  const D3DVALUE& operator()(int iRow, int iColumn) const { return m[iRow][iColumn]; }
#endif
} D3DMATRIX, *LPD3DMATRIX;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

See Also

D3DMATRIX

Division Operator (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. Both scalar and vector forms of this operator have been implemented. The division operator is defined as follows:

```
_D3DVECTOR operator / (const _D3DVECTOR& v, D3DVALUE s);
_D3DVECTOR operator / (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline _D3DVECTOR
operator / (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return _D3DVECTOR(v1.x/v2.x, v1.y/v2.y, v1.z/v2.z);
}

inline _D3DVECTOR
operator / (const _D3DVECTOR& v, D3DVALUE s)
{
  return _D3DVECTOR(v.x/s, v.y/s, v.z/s);
```

```
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

Multiplication Operator (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. Both scalar and vector forms of this operator have been implemented. The multiplication operator is defined as follows:

```
_D3DVECTOR operator * (const _D3DVECTOR& v, D3DVALUE s);
_D3DVECTOR operator * (D3DVALUE s, const _D3DVECTOR& v);
_D3DVECTOR operator * (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline _D3DVECTOR
operator * (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return _D3DVECTOR(v1.x*v2.x, v1.y*v2.y, v1.z*v2.z);
}

inline _D3DVECTOR
operator * (const _D3DVECTOR& v, D3DVALUE s)
{
  return _D3DVECTOR(s*v.x, s*v.y, s*v.z);
}

inline _D3DVECTOR
operator * (D3DVALUE s, const _D3DVECTOR& v)
{
  return _D3DVECTOR(s*v.x, s*v.y, s*v.z);
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

Subtraction Operator (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. The subtraction operator is defined as follows:

```
_D3DVECTOR operator - (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline _D3DVECTOR
operator - (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return _D3DVECTOR(v1.x-v2.x, v1.y-v2.y, v1.z-v2.z);
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

Unary Operators (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

The unary operators are overloaded operators for the D3D_OVERLOADS extensions. The unary operators are defined as follows:

```
_D3DVECTOR operator + (const _D3DVECTOR& v);
_D3DVECTOR operator - (const _D3DVECTOR& v);

inline _D3DVECTOR
operator + (const _D3DVECTOR& v)
{
  return v;
}

inline _D3DVECTOR
operator - (const _D3DVECTOR& v)
{
  return _D3DVECTOR(-v.x, -v.y, -v.z);
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

Vector Dominance Operators (D3D_OVERLOADS)

[This is preliminary documentation and subject to change.]

These binary operators are overloaded operators for the D3D_OVERLOADS extensions. Vector v1 dominates vector v2 if any component of v1 is greater than the corresponding component of v2. Therefore, it is possible for neither of the two specified vectors to dominate the other.

```
int operator < (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
int operator <= (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

The vector-dominance operators are defined as follows:

```
inline int
operator < (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return v1[0] < v2[0] && v1[1] < v2[1] && v1[2] < v2[2];
}

inline int
operator <= (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return v1[0] <= v2[0] && v1[1] <= v2[1] && v1[2] <= v2[2];
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3D_OVERLOADS Helper Functions

[This is preliminary documentation and subject to change.]

This section contains reference information for the helper functions provided by the D3D_OVERLOADS C++ extensions.

- **CrossProduct**
- **DotProduct**
- **Magnitude**
- **Max**
- **Maximize**
- **Min**
- **Minimize**
- **Normalize**
- **SquareMagnitude**

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

CrossProduct

[This is preliminary documentation and subject to change.]

This helper function returns the cross product of the specified vectors. **CrossProduct** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR CrossProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline _D3DVECTOR
CrossProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
    _D3DVECTOR result;

    result[0] = v1[1] * v2[2] - v1[2] * v2[1];
    result[1] = v1[2] * v2[0] - v1[0] * v2[2];
    result[2] = v1[0] * v2[1] - v1[1] * v2[0];

    return result;
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

See Also

DotProduct

DotProduct

[This is preliminary documentation and subject to change.]

This helper function returns the dot product of the specified vectors. **DotProduct** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE DotProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline D3DVALUE
DotProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return v1.x*v2.x + v1.y * v2.y + v1.z*v2.z;
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

See Also

CrossProduct

Magnitude

[This is preliminary documentation and subject to change.]

This helper function returns the absolute value of the specified vector. **Magnitude** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE Magnitude (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE
Magnitude (const _D3DVECTOR& v)
{
  return (D3DVALUE) sqrt(SquareMagnitude(v));
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

See Also

SquareMagnitude

Max

[This is preliminary documentation and subject to change.]

This helper function returns the maximum component of the specified vector. **Max** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE Max (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE
Max (const _D3DVECTOR& v)
{
  D3DVALUE ret = v.x;
  if (ret < v.y) ret = v.y;
  if (ret < v.z) ret = v.z;
  return ret;
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

See Also

Min

Maximize

[This is preliminary documentation and subject to change.]

This helper function returns a vector that is made up of the largest components of the two specified vectors. **Maximize** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR Maximize (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline _D3DVECTOR
Maximize (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return _D3DVECTOR( v1[0] > v2[0] ? v1[0] : v2[0],
                     v1[1] > v2[1] ? v1[1] : v2[1],
                     v1[2] > v2[2] ? v1[2] : v2[2]);
}
```

Remarks

You could use the **Maximize** and **Minimize** functions to compute the bounding box for a set of points, in a function that looks like this:

```
   void
   ComputeBoundingBox(const D3DVECTOR *pts, int N, D3DVECTOR *min, D3DVECTOR
*max)
   {
      int i;
      *min = *max = pts[0];
      for (i = 1; i < N; i += 1)
      {
         *min = Minimize(*min, pts[i]);
         *max = Maximize(*max, pts[i]);
      }
   }
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

See Also

Minimize

Min

[This is preliminary documentation and subject to change.]

This helper function returns the minimum component of the specified vector. **Min** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE Min (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE
Min (const _D3DVECTOR& v)
{
  D3DVALUE ret = v.x;
  if (v.y < ret) ret = v.y;
  if (v.z < ret) ret = v.z;
  return ret;
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

See Also

Max

Minimize

[This is preliminary documentation and subject to change.]

This helper function returns a vector that is made up of the smallest components of the two specified vectors. **Minimize** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR Minimize (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline _D3DVECTOR
```

```
Minimize (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
  return _D3DVECTOR( v1[0] < v2[0] ? v1[0] : v2[0],
                     v1[1] < v2[1] ? v1[1] : v2[1],
                     v1[2] < v2[2] ? v1[2] : v2[2]);
}
```

Remarks

You could use the **Maximize** and **Minimize** functions to compute the bounding box for a set of points, in a function that looks like this:

```
   void
   ComputeBoundingBox(const D3DVECTOR *pts, int N, D3DVECTOR *min, D3DVECTOR
*max)
   {
      int i;
      *min = *max = pts[0];
      for (i = 1; i < N; i += 1)
      {
         *min = Minimize(*min, pts[i]);
         *max = Maximize(*max, pts[i]);
      }
   }
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

See Also

Maximize

Normalize

[This is preliminary documentation and subject to change.]

This helper function returns the normalized version of the specified vector (that is, a unit-length vector with the same direction as the source). **Normalize** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR Normalize (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline _D3DVECTOR
Normalize (const _D3DVECTOR& v)
{
  return v / Magnitude(v);
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

SquareMagnitude

[This is preliminary documentation and subject to change.]

This helper function returns the square of the absolute value of the specified vector. **SquareMagnitude** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE SquareMagnitude (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE
SquareMagnitude (const _D3DVECTOR& v)
{
  return v.x*v.x + v.y*v.y + v.z*v.z;
}
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3d.h.

See Also

Magnitude

Callback Functions

[This is preliminary documentation and subject to change.]

This section contains reference information for the callback functions you may need to implement when you work with Direct3D Immediate Mode.

- **D3DEnumDevicesCallback**
- **D3DEnumPixelFormatsCallback**
- **D3DEnumTextureFormatsCallback**
- **D3DValidateCallback**

D3DEnumDevicesCallback

[This is preliminary documentation and subject to change.]

The **D3DEnumDevicesCallback** is an application-defined callback function for the **IDirect3D3::EnumDevices** method.

```
HRESULT CALLBACK D3DEnumDevicesCallback(
  GUID FAR* lpGuid,
  LPSTR lpDeviceDescription,
  LPSTR lpDeviceName,
  LPD3DDEVICEDESC lpD3DHWDeviceDesc,
  LPD3DDEVICEDESC lpD3DHELDeviceDesc,
  LPVOID lpContext
);
```

Parameters

lpGuid
Address of a globally unique identifier (GUID) that identifies a Direct3D device.

lpDeviceDescription
Address of a textual description of the device.

lpDeviceName
Address of the device name.

lpD3DHWDeviceDesc
Address of a **D3DDEVICEDESC** structure that contains the hardware capabilities of the Direct3D device.

lpD3DHELDeviceDesc
Address of a **D3DDEVICEDESC** structure that contains the emulated capabilities of the Direct3D device.

lpContext
Address of application-defined data passed to this callback function.

Return Values

Applications should return D3DENUMRET_OK to continue the enumeration, or D3DENUMRET_CANCEL to cancel it.

Remarks

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

The **LPD3DENUMDEVICESCALLBACK** data type is defined as a pointer to this callback function:

```
typedef HRESULT (FAR PASCAL * LPD3DENUMDEVICESCALLBACK)(
  GUID FAR *lpGuid, LPSTR lpDeviceDescription, LPSTR lpDeviceName,
  LPD3DDEVICEDESC, LPD3DDEVICEDESC, LPVOID);
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dcaps.h.
Import Library: User-defined.

D3DEnumPixelFormatsCallback

[This is preliminary documentation and subject to change.]

The **D3DEnumPixelFormatsCallback** is an application-defined callback function for the **IDirect3D3::EnumZBufferFormats**, and **IDirect3DDevice3::EnumTextureFormats** methods.

```
HRESULT CALLBACK D3DEnumPixelFormatsCallback(
  LPDDPIXELFORMAT lpDDPixFmt,
  LPVOID          lpContext
);
```

Parameters

lpDDPixFmt
Address of a **DDPIXELFORMAT** structure that describes the enumerated pixel format.

lpContext
Address of application-defined data passed to the function by the caller.

Return Values

Applications should return D3DENUMRET_OK to continue the enumeration, or D3DENUMRET_CANCEL to cancel it.

Remarks

The **LPD3DENUMPIXELFORMATSCALLBACK** data type is defined as a pointer to this callback function:

```
typedef HRESULT (WINAPI* LPD3DENUMPIXELFORMATSCALLBACK)
(LPDDPIXELFORMAT lpDDPixFmt, LPVOID lpContext);
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dcaps.h.
Import Library: User-defined.

D3DEnumTextureFormatsCallback

[This is preliminary documentation and subject to change.]

The **D3DEnumTextureFormatsCallback** function is an application-defined callback function for the **IDirect3DDevice::EnumTextureFormats** and **IDirect3DDevice2::EnumTextureFormats** methods.

```
HRESULT CALLBACK D3DEnumTextureFormatsCallback(
  LPDDSURFACEDESC lpDdsd,
  LPVOID lpUserArg
);
```

Parameters

lpDdsd
Address of a **DDSURFACEDESC** structure containing the texture information.

lpUserArg
Address of application-defined data passed to this callback function.

Return Values

Applications should return D3DENUMRET_OK to continue the enumeration, or D3DENUMRET_CANCEL to cancel it.

Remarks

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

The **LPD3DENUMTEXTUREFORMATSCALLBACK** data type is defined as a pointer to this callback function:

```
typedef HRESULT (WINAPI* LPD3DENUMTEXTUREFORMATSCALLBACK)
(LPDDSURFACEDESC lpDdsd, LPVOID lpContext);
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.
Import Library: User-defined.

D3DValidateCallback

[This is preliminary documentation and subject to change.]

The **D3DValidateCallback** function is an application-defined callback function for the **IDirect3DExecuteBuffer::Validate** method. The **IDirect3DExecuteBuffer::Validate** method is not currently implemented.

IDirect3DExecuteBuffer::Validate is a debugging routine that checks the execute buffer and returns an offset into the buffer when any errors are encountered.

```
HRESULT CALLBACK D3DValidateCallback(
  LPVOID lpUserArg,
  DWORD dwOffset
);
```

Parameters

lpUserArg
Address of application-defined data passed to this callback function.

dwOffset
Offset into the execute buffer at which the system found an error.

Return Values

Applications should return D3DENUMRET_OK to continue the enumeration, or D3DENUMRET_CANCEL to cancel it.

Remarks

When determining the order in which to call callback functions, the system searches the objects highest in the hierarchy first, and then calls their callback functions in the order in which they were created.

The **LPD3DVALIDATECALLBACK** data type is defined as a pointer to this callback function:

```
typedef HRESULT (WINAPI* LPD3DVALIDATECALLBACK)(LPVOID lpUserArg, DWORD
dwOffset);
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.
Import Library: User-defined.

Macros

[This is preliminary documentation and subject to change.]

This section contains reference information for the macros provided by Direct3D Immediate Mode.

- **D3DDivide**
- **D3DMultiply**
- **D3DRGB**
- **D3DRGBA**
- **D3DSTATE_OVERRIDE**
- **D3DVAL**
- **D3DVALP**
- **RGB_GETBLUE**
- **RGB_GETGREEN**
- **RGB_GETRED**
- **RGB_MAKE**
- **RGB_TORGBA**
- **RGBA_GETALPHA**
- **RGBA_GETBLUE**
- **RGBA_GETGREEN**
- **RGBA_GETRED**
- **RGBA_MAKE**
- **RGBA_SETALPHA**
- **RGBA_TORGB**

D3DDivide

[This is preliminary documentation and subject to change.]

The **D3DDivide** macro divides two values.

```
D3DDivide(a, b)    (float)((double) (a) / (double) (b))
```

Parameters

a and *b*
Dividend and divisor in the expression, respectively.

Return Values

The macros returns the quotient of the division.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DMultiply

D3DMultiply

[This is preliminary documentation and subject to change.]

The **D3DMultiply** macro multiplies two values.

```
D3DMultiply(a, b)    ((a) * (b))
```

Parameters

a and *b*
Values to be multiplied.

Return Values

The macros returns the product of the multiplication.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DDivide

D3DRGB

[This is preliminary documentation and subject to change.]

The **D3DRGB** macro initializes a color with the supplied RGB values.

```
D3DRGB(r, g, b) \
    (0xff000000L | ( ((long)((r) * 255)) << 16) | \
    (((long)((g) * 255)) << 8) | (long)((b) * 255))
```

Parameters

r, *g*, and *b*
Red, green, and blue components of the color. These should be floating-point values in the range 0 through 1.

Return Values

The macros returns the **D3DCOLOR** value corresponding to the supplied RGB values.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DRGBA

D3DRGBA

[This is preliminary documentation and subject to change.]

The **D3DRGBA** macro initializes a color with the supplied RGBA values.

```
D3DRGBA(r, g, b, a) \
    ((((long)((a) * 255)) << 24) | (((long)((r) * 255)) << 16) |
    (((long)((g) * 255)) << 8) | (long)((b) * 255))
```

Parameters

r, *g*, *b*, and *a*
Red, green, blue, and alpha components of the color.

Return Values

The macros returns the **D3DCOLOR** value corresponding to the supplied RGBA values.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DRGB

D3DSTATE_OVERRIDE

[This is preliminary documentation and subject to change.]

The **D3DSTATE_OVERRIDE** macro overrides the state of the rasterization, lighting, or transformation module. Applications can use this macro to lock and unlock a state.

```
D3DSTATE_OVERRIDE(type) (D3DRENDERSTATETYPE)(((DWORD) (type) +
D3DSTATE_OVERRIDE_BIAS))
```

Parameters

type
State to override. This parameter should be one of the members of the **D3DTRANSFORMSTATETYPE**, **D3DLIGHTSTATETYPE**, or **D3DRENDERSTATETYPE** enumerated types.

Return Values

No return value.

Remarks

An application might, for example, use the **STATE_DATA** helper macro (defined in the D3dmacs.h header file in the Misc directory of the DirectX Programmer's Reference sample code) and **D3DSTATE_OVERRIDE** to lock and unlock the D3DRENDERSTATE_SHADEMODE render state:

```
// Lock the shade mode.

STATE_DATA(D3DSTATE_OVERRIDE(D3DRENDERSTATE_SHADEMODE), TRUE,
lpBuffer);

// Work with the shade mode and unlock it when read-only status is not required.

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

For more information about overriding rendering states, see States and State Overrides.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DVAL

[This is preliminary documentation and subject to change.]

The **D3DVAL** macro creates a value whose type is **D3DVALUE**.

```
D3DVAL(val)    ((float)val)
```

Parameters

val
Value to be converted.

Return Values

The macros returns the converted value.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DVALP

D3DVALP

[This is preliminary documentation and subject to change.]

The **D3DVALP** macro creates a value of the specified precision.

```
D3DVALP(val, prec)    ((float)val)
```

Parameters

val
Value to be converted.

prec
Ignored.

Return Values

The macros returns the converted value.

Remarks

The precision, as implemented by the **D3DVAL** macro, is 16 bits for the fractional part of the value.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DVAL

RGB_GETBLUE

[This is preliminary documentation and subject to change.]

The **RGB_GETBLUE** macro retrieves the blue component of a **D3DCOLOR** value.

```
RGB_GETBLUE(rgb)    ((rgb) & 0xff)
```

Parameters

rgb
Color index from which the blue component is retrieved.

Return Values

Returns the blue component.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

RGB_GETGREEN

[This is preliminary documentation and subject to change.]

The **RGB_GETGREEN** macro retrieves the green component of a **D3DCOLOR** value.

```
RGB_GETGREEN(rgb)    (((rgb) >> 8) & 0xff)
```

Parameters

rgb
Color index from which the green component is retrieved.

Return Values

The macros returns the green component.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.
Header: Declared in d3dtypes.h.

RGB_GETRED

[This is preliminary documentation and subject to change.]

The **RGB_GETRED** macro retrieves the red component of a **D3DCOLOR** value.

```
RGB_GETRED(rgb)    (((rgb) >> 16) & 0xff)
```

Parameters

rgb
Color index from which the red component is retrieved.

Return Values

The macros returns the red component.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

RGB_MAKE

[This is preliminary documentation and subject to change.]

The **RGB_MAKE** macro creates an RGB color from supplied values.

```
RGB_MAKE(r, g, b)    ((D3DCOLOR) (((r) << 16) | ((g) << 8) | (b)))
```

Parameters

r, *g*, and *b*
Red, green, and blue components of the color to be created. These should be integer values in the range zero through 255.

Return Values

The macros returns the color.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

RGB_TORGBA

[This is preliminary documentation and subject to change.]

The **RGB_TORGBA** macro creates an RGBA color from a supplied RGB color.

```
RGB_TORGBA(rgb)    ((D3DCOLOR) ((rgb) | 0xff000000))
```

Parameters

rgb
RGB color to be converted to an RGBA color.

Return Values

Returns the RGBA color.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

RGBA_TORGB

RGBA_GETALPHA

[This is preliminary documentation and subject to change.]

The **RGBA_GETALPHA** macro retrieves the alpha component of an RGBA **D3DCOLOR** value.

```
RGBA_GETALPHA(rgb)    ((rgb) >> 24)
```

Parameters

rgb

Color index from which the alpha component is retrieved.

Return Values

The macros returns the alpha component.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

RGBA_GETBLUE

[This is preliminary documentation and subject to change.]

The **RGBA_GETBLUE** macro retrieves the blue component of an RGBA **D3DCOLOR** value.

```
RGBA_GETBLUE(rgb)    ((rgb) & 0xff)
```

Parameters

rgb
Color index from which the blue component is retrieved.

Return Values

The macros returns the blue component.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

RGBA_GETGREEN

[This is preliminary documentation and subject to change.]

The **RGBA_GETGREEN** macro retrieves the green component of an RGBA **D3DCOLOR** value.

```
RGBA_GETGREEN(rgb)    (((rgb) >> 8) & 0xff)
```

Parameters

rgb
Color index from which the green component is retrieved.

Return Values

The macros returns the green component.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

RGBA_GETRED

[This is preliminary documentation and subject to change.]

The **RGBA_GETRED** macro retrieves the red component of an RGBA **D3DCOLOR** value.

```
RGBA_GETRED(rgb)    (((rgb) >> 16) & 0xff)
```

Parameters

rgb
Color index from which the red component is retrieved.

Return Values

The macros returns the red component.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

RGBA_MAKE

[This is preliminary documentation and subject to change.]

The **RGBA_MAKE** macro creates an RGBA **D3DCOLOR** value from supplied red, green, blue, and alpha components.

```
RGBA_MAKE(r, g, b, a)   \
    ((D3DCOLOR) (((a) << 24) | ((r) << 16) | ((g) << 8) | (b)))
```

Parameters

r, *g*, *b*, and *a*
Red, green, blue, and alpha components of the RGBA color to be created.

Return Values

The macros returns the color.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

RGBA_SETALPHA

[This is preliminary documentation and subject to change.]

The **RGBA_SETALPHA** macro sets the alpha component of an RGBA **D3DCOLOR** value.

```
RGBA_SETALPHA(rgba, x)    (((x) << 24) | ((rgba) & 0x00ffffff))
```

Parameters

rgba
RGBA color for which the alpha component will be set.

x
Value of alpha component to be set.

Return Values

The macros returns the RGBA color whose alpha component has been set.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

RGBA_TORGB

[This is preliminary documentation and subject to change.]

The **RGBA_TORGB** macro creates an RGB **D3DCOLOR** value from a supplied RGBA **D3DCOLOR** value by stripping off the alpha component of the color.

```
RGBA_TORGB(rgba)    ((D3DCOLOR) ((rgba) & 0xffffff))
```

Parameters

rgba
RGBA color to be converted to an RGB color.

Return Values

The macros returns the RGB color.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

RGB_TORGBA

Structures

[This is preliminary documentation and subject to change.]

This section contains information about the following structures used with Direct3D Immediate Mode.

- **D3DBRANCH**
- **D3DCLIPSTATUS**
- **D3DCOLORVALUE**
- **D3DDEVICEDESC**
- **D3DDP_PTRSTRIDE**
- **D3DDRAWPRIMITIVESTRIDEDDATA**
- **D3DEXECUTEBUFFERDESC**
- **D3DEXECUTEDATA**
- **D3DFINDDEVICERESULT**

- **D3DFINDDEVICESEARCH**
- **D3DHVERTEX**
- **D3DINSTRUCTION**
- **D3DLIGHT**
- **D3DLIGHT2**
- **D3DLIGHTDATA**
- **D3DLIGHTINGCAPS**
- **D3DLIGHTINGELEMENT**
- **D3DLINE**
- **D3DLINEPATTERN**
- **D3DLVERTEX**
- **D3DMATERIAL**
- **D3DMATRIX**
- **D3DMATRIXLOAD**
- **D3DMATRIXMULTIPLY**
- **D3DPICKRECORD**
- **D3DPOINT**
- **D3DPRIMCAPS**
- **D3DPROCESSVERTICES**
- **D3DRECT**
- **D3DSPAN**
- **D3DSTATE**
- **D3DSTATS**
- **D3DSTATUS**
- **D3DTEXTURELOAD**
- **D3DTLVERTEX**
- **D3DTRANSFORMCAPS**
- **D3DTRANSFORMDATA**
- **D3DTRIANGLE**
- **D3DVECTOR**
- **D3DVERTEX**
- **D3DVERTEXBUFFERDESC**
- **D3DVIEWPORT**
- **D3DVIEWPORT2**

Note

The memory for all DirectX structures must be initialized to zero before use. In addition, all structures that contain a **dwSize** member must set the member to the size of the structure, in bytes, before use. The following example performs these tasks on a common structure, **DDCAPS**:

```
DDCAPS ddcaps; // Can't use this yet.

ZeroMemory(&ddcaps, sizeof(ddcaps));
ddcaps.dwSize = sizeof(ddcaps);

// Now the structure can be used.
.
.
```

D3DBRANCH

[This is preliminary documentation and subject to change.]

The **D3DBRANCH** structure performs conditional operations inside an execute buffer. This structure is a forward-branch structure.

```
typedef struct _D3DBRANCH {
    DWORD dwMask;
    DWORD dwValue;
    BOOL  bNegate;
    DWORD dwOffset;
} D3DBRANCH, *LPD3DBRANCH;
```

Members

dwMask
Bitmask for the branch. This mask is combined with the driver-status mask by using the bitwise **AND** operator. If the result equals the value specified in the **dwValue** member and the **bNegate** member is FALSE, the branch is taken.

For a list of the available driver-status masks, see the **dwStatus** member of the **D3DSTATUS** structure.

dwValue
Application-defined value to compare against the operation described in the **dwMask** member.

bNegate
TRUE to negate comparison.

dwOffset
How far to branch forward. Specify zero to exit.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DCLIPSTATUS

[This is preliminary documentation and subject to change.]

The **D3DCLIPSTATUS** structure describes the current clip status and extents of the clipping region. This structure was introduced in DirectX 5.0.

```
typedef struct _D3DCLIPSTATUS {
    DWORD dwFlags;
    DWORD dwStatus;
    float minx, maxx;
    float miny, maxy;
    float minz, maxz;
} D3DCLIPSTATUS, *LPD3DCLIPSTATUS;
```

Members

dwFlags
Flags describing whether this structure describes 2-D extents, 3-D extents, or the clip status. This member can be a combination of the following flags:

D3DCLIPSTATUS_STATUS
The structure describes the current clip status.

D3DCLIPSTATUS_EXTENTS2
The structure describes the current 2-D extents. This flag cannot be combined with D3DCLIPSTATUS_EXTENTS3.

D3DCLIPSTATUS_EXTENTS3
Not currently implemented.

dwStatus
Describes the current clip status. For a list of the available driver-status masks, see the **dwStatus** member of the **D3DSTATUS** structure.

minx, maxx, miny, maxy, minz, maxz
x, y, and z extents of the current clipping region.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice3::GetClipStatus, **IDirect3DDevice3::SetClipStatus**

D3DCOLORVALUE

[This is preliminary documentation and subject to change.]

The **D3DCOLORVALUE** structure describes color values for the **D3DLIGHT2** and **D3DMATERIAL** structures.

```
typedef struct _D3DCOLORVALUE {
    union {
        D3DVALUE r;
        D3DVALUE dvR;
    };
    union {
        D3DVALUE g;
        D3DVALUE dvG;
    };
    union {
        D3DVALUE b;
        D3DVALUE dvB;
    };
    union {
        D3DVALUE a;
        D3DVALUE dvA;
    };
} D3DCOLORVALUE;
```

Members

dvR, **dvG**, **dvB**, and **dvA**
Values of the **D3DVALUE** type specifying the red, green, blue, and alpha components of a color. These values generally range from 0 to 1, with 0 being black.

Remarks

You can set the members of this structure to values outside the range of 0 to 1 to implement some unusual effects. Values greater than 1 produce strong lights that tend to wash out a scene. Negative values produce dark lights, which actually remove light from a scene. For more information, see Colored Lights.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DDEVICEDESC

[This is preliminary documentation and subject to change.]

The **D3DDEVICEDESC** structure contains a description of the current device. This structure is used to query the current device by such methods as **IDirect3DDevice3::GetCaps**.

```
typedef struct _D3DDeviceDesc {
    DWORD           dwSize;
    DWORD           dwFlags;
    D3DCOLORMODEL   dcmColorModel;
    DWORD           dwDevCaps;
    D3DTRANSFORMCAPS dtcTransformCaps;
    BOOL            bClipping;
    D3DLIGHTINGCAPS  dlcLightingCaps;
    D3DPRIMCAPS     dpcLineCaps;
    D3DPRIMCAPS     dpcTriCaps;
    DWORD           dwDeviceRenderBitDepth;
    DWORD           dwDeviceZBufferBitDepth;
    DWORD           dwMaxBufferSize;
    DWORD           dwMaxVertexCount;
    DWORD           dwMinTextureWidth, dwMinTextureHeight;
    DWORD           dwMaxTextureWidth, dwMaxTextureHeight;
    DWORD           dwMinStippleWidth, dwMaxStippleWidth;
    DWORD           dwMinStippleHeight, dwMaxStippleHeight;
    DWORD           dwMaxTextureRepeat;
    DWORD           dwMaxTextureAspectRatio;
    DWORD           dwMaxAnisotropy;
    D3DVALUE        dvGuardBandLeft;
    D3DVALUE        dvGuardBandTop;
    D3DVALUE        dvGuardBandRight;
    D3DVALUE        dvGuardBandBottom;
    D3DVALUE        dvExtentsAdjust;
    DWORD           dwStencilCaps;
    DWORD           dwFVFCaps;
    DWORD           dwTextureOpCaps;
    WORD            wMaxTextureBlendStages;
    WORD            wMaxSimultaneousTextures;
} D3DDEVICEDESC, *LPD3DDEVICEDESC;
```

Members

dwSize
Size, in bytes, of this structure. You can use the **D3DDEVICEDESCSIZE** constant for this value. This member must be initialized before the structure is used.

dwFlags
Flags identifying the members of this structure that contain valid data.

D3DDD_BCLIPPING
The **bClipping** member is valid.

D3DDD_COLORMODEL
The **dcmColorModel** member is valid.

D3DDD_DEVCAPS
The **dwDevCaps** member is valid.

D3DDD_DEVICERENDERBITDEPTH
The **dwDeviceRenderBitDepth** member is valid.

D3DDD_DEVICEZBUFFERBITDEPTH
The **dwDeviceZBufferBitDepth** member is valid.

D3DDD_LIGHTINGCAPS
The **dlcLightingCaps** member is valid.

D3DDD_LINECAPS
The **dpcLineCaps** member is valid.

D3DDD_MAXBUFFERSIZE
The **dwMaxBufferSize** member is valid.

D3DDD_MAXVERTEXCOUNT
The **dwMaxVertexCount** member is valid.

D3DDD_TRANSFORMCAPS
The **dtcTransformCaps** member is valid.

D3DDD_TRICAPS
The **dpcTriCaps** member is valid.

dcmColorModel
One of the values of the **D3DCOLORMODEL** data type, specifying the color model for the device.

dwDevCaps
Flags identifying the capabilities of the device.

D3DDEVCAPS_CANRENDERAFTERFLIP
Device can queue rendering commands after a page flip. Applications should not change their behavior if this flag is set; this capability simply means that the device is relatively fast.

This flag was introduced in DirectX 5.0.

D3DDEVCAPS_DRAWPRIMTLVERTEX
Device exports a DrawPrimitive-aware HAL.

This flag was introduced in DirectX 5.0.

D3DDEVCAPS_EXECUTESYSTEMMEMORY
Device can use execute buffers from system memory.

D3DDEVCAPS_EXECUTEVIDEOMEMORY
Device can use execute buffer from video memory.

D3DDEVCAPS_FLOATTLVERTEX
Device accepts floating point for post-transform vertex data.

D3DDEVCAPS_SEPARATETEXTUREMEMORIES
Device uses discrete texture memory pools for each stage. Textures must be assigned to texture stages explicitly at creation-time by setting the **dwTextureStage** member of the **DDSURFACEDESC2** structure to the appropriate stage identifier.

D3DDEVCAPS_SORTDECREASINGZ
Device needs data sorted for decreasing depth.

D3DDEVCAPS_SORTEXACT
Device needs data sorted exactly.

D3DDEVCAPS_SORTINCREASINGZ
Device needs data sorted for increasing depth.

D3DDEVCAPS_TEXREPEATNOTSCALEDBYSIZE
Device defers scaling of texture indices by the texture size until after the texture address mode is applied.

D3DDEVCAPS_TEXTURENONLOCALVIDMEM
Device can retrieve textures from non-local video (AGP) memory.

This flag was introduced in DirectX 5.0. For more information about AGP memory, see Using Non-local Video Memory Surfaces in the DirectDraw documentation.

D3DDEVCAPS_TEXTURESYSTEMMEMORY
Device can retrieve textures from system memory.

D3DDEVCAPS_TEXTUREVIDEOMEMORY
Device can retrieve textures from device memory.

D3DDEVCAPS_TLVERTEXSYSTEMMEMORY
Device can use buffers from system memory for transformed and lit vertices.

D3DDEVCAPS_TLVERTEXVIDEOMEMORY
Device can use buffers from video memory for transformed and lit vertices.

dtcTransformCaps
One of the members of the **D3DTRANSFORMCAPS** structure, specifying the transformation capabilities of the device.

bClipping
TRUE if the device can perform 3-D clipping.

dlcLightingCaps
One of the members of the **D3DLIGHTINGCAPS** structure, specifying the lighting capabilities of the device.

dpcLineCaps and **dpcTriCaps**

D3DPRIMCAPS structures defining the device's support for line-drawing and triangle primitives.

dwDeviceRenderBitDepth
Device's rendering bit-depth. This can be one or more of the following DirectDraw bit-depth constants: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

dwDeviceZBufferBitDepth
Device's depth-buffer bit-depth. This can be one of the following DirectDraw bit-depth constants: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

dwMaxBufferSize
Maximum size of the execute buffer for this device. If this member is 0, the application can use any size.

dwMaxVertexCount
Maximum number of vertices supported by an execute buffer for this device. (The DrawPrimitive rendering methods support up to 65,536 vertices.)

dwMinTextureWidth, dwMinTextureHeight
Minimum texture width and height for this device.

dwMaxTextureWidth, dwMaxTextureHeight
Maximum texture width and height for this device.

dwMinStippleWidth, dwMaxStippleWidth
Minimum and maximum width of the stipple pattern for this device.

dwMinStippleHeight, dwMaxStippleHeight
Minimum and maximum height of the stipple pattern for this device.

dwMaxTextureRepeat
Full range of the integer (non-fractional) bits of the post-normalized texture indices. If the D3DDEVCAPS_TEXREPEATNOTSCALEDBYSIZE bit is set, the device defers scaling by the texture size until after the texture address mode is applied. If it isn't set, the device scales the texture indices by the texture size (largest level-of-detail) prior to interpolation.

dwMaxTextureAspectRatio
Maximum texture aspect ratio supported by the hardware; this will typically be a power of 2.

dwMaxAnisotropy
Maximum valid value for the D3DRENDERSTATE_ANISOTROPY render state.

dvGuardBandLeft, **dvGuardBandTop**, **dvGuardBandRight**, and **dvGuardBandBottom**
The screen-space coordinates of the guard-band clipping region. Coordinates inside this rectangle but outside the viewport rectangle will automatically be clipped.

dvExtentsAdjust
Number of pixels to adjust the extents rectangle outward to accommodate antialiasing kernels.

dwStencilCaps

Flags specifying supported stencil-buffer operations. Stencil operations are assumed to be valid for all three stencil-buffer operation render states (D3DRENDERSTATE_STENCILFAIL, D3DRENDERSTATE_STENCILPASS, and D3DRENDERSTATE_STENCILFAILZFAIL).

D3DSTENCILCAPS_DECR
The **D3DSTENCILOP_DECR** operation is supported.

D3DSTENCILCAPS_DECRSAT
The **D3DSTENCILOP_DECRSAT** operation is supported.

D3DSTENCILCAPS_INCR
The **D3DSTENCILOP_INCR** operation is supported.

D3DSTENCILCAPS_INCRSAT
The **D3DSTENCILOP_INCRSAT** operation is supported.

D3DSTENCILCAPS_INVERT
The **D3DSTENCILOP_INVERT** operation is supported.

D3DSTENCILCAPS_KEEP
The **D3DSTENCILOP_KEEP** operation is supported.

D3DSTENCILCAPS_REPLACE
The **D3DSTENCILOP_REPLACE** operation is supported.

D3DSTENCILCAPS_ZERO
The **D3DSTENCILOP_ZERO** operation is supported.

dwFVFCaps
Flexible vertex format capabilities:

D3DFVFCAPS_DONOTSTRIPELEMENTS
Device prefers that vertex elements not be stripped. That is, if the vertex format contains elements that will not be used with the current render states, there is no need to regenerate the vertices. If this capability flag is not present, stripping extraneous elements from the vertex format will provide better performance.

D3DFVFCAPS_TEXCOORDCOUNTMASK
Masks the low WORD of **dwFVFCaps**. These bits, cast to the WORD data type, describe the total number of texture coordinate sets that the device can simultaneously use for multiple texture blending. (You can use up to eight texture coordinate sets for any vertex, but the device can only blend using the specified number of texture coordinate sets.)

dwTextureOpCaps
Combination of flags describing the texture operations supported by this device. The following flags are defined:

D3DTEXOPCAPS_ADD
The **D3DTOP_ADD** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSIGNED
The **D3DTOP_ADDSIGNED** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSIGNED2X

The **D3DTOP_ADDSIGNED2X** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSMOOTH
The **D3DTOP_ADDSMOOTH** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDCURRENTALPHA
The **D3DTOP_BLENDCURRENTALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDDIFFUSEALPHA
The **D3DTOP_BLENDDIFFUSEALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDFACTORALPHA
The **D3DTOP_BLENDFACTORALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDTEXTUREALPHA
The **D3DTOP_BLENDTEXTUREALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDTEXTUREALPHAPM
The **D3DTOP_BLENDTEXTUREALPHAPM** texture blending operation is supported by this device.

D3DTEXOPCAPS_BUMPENVMAP
The **D3DTOP_BUMPENVMAP** texture blending operation is supported by this device.

D3DTEXOPCAPS_BUMPENVMAPLUMINANCE
The **D3DTOP_BUMPENVMAPLUMINANCE** texture blending operation is supported by this device.

D3DTEXOPCAPS_DISABLE
The **D3DTOP_DISABLE** texture blending operation is supported by this device.

D3DTEXOPCAPS_DOTPRODUCT3
The **D3DTOP_DOTPRODUCT3** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE
The **D3DTOP_MODULATE** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE2X
The **D3DTOP_MODULATE2X** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE4X
The **D3DTOP_MODULATE4X** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR
The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA
The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR
The **D3DTOP_MODULATEINVALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA
The **D3DTOP_MODULATEINVCOLOR_ADDALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_PREMODULATE
The **D3DTOP_PREMODULATE** texture blending operation is supported by this device.

D3DTEXOPCAPS_SELECTARG1
The **D3DTOP_SELECTARG1** texture blending operation is supported by this device.

D3DTEXOPCAPS_SELECTARG2
The **D3DTOP_SELECTARG2** texture blending operation is supported by this device.

D3DTEXOPCAPS_SUBTRACT
The **D3DTOP_SUBTRACT** texture blending operation is supported by this device.

wMaxTextureBlendStages
Maximum number of texture blending stages supported by this device.

wMaxSimultaneousTextures
Maximum number of textures that can be simultaneously bound to the texture blending stages for this device. See remarks.

Remarks

The **wMaxTextureBlendStages** and **wMaxSimultaneousTextures** members might seem very similar at first glance, but they contain different information. The **wMaxTextureBlendStages** member contains the total number of texture-blending stages supported by the current device, and the **wMaxSimultaneousTextures** member describes how many of those stages can have textures bound to them by using the **IDirect3DDevice3::SetTexture** method.

When the driver fills this structure, it can set values for execute buffer capabilities even when the interface being used to retrieve the capabilities (such as **IDirect3DDevice3**) doesn't support execute buffers.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dcaps.h.

See Also

D3DCOLORMODEL, **D3DFINDDEVICERESULT**, **D3DLIGHTINGCAPS**, **D3DPRIMCAPS**, **D3DTRANSFORMCAPS**

D3DDP_PTRSTRIDE

[This is preliminary documentation and subject to change.]

The **D3DDP_PTRSTRIDE** structure contains the address of an array of flexible vertex format components and the stride to the next element in the array. This structure is contained by the **D3DDRAWPRIMITIVESTRIDEDDATA** structure.

```
typedef struct _D3DDP_PTRSTRIDE {
    LPVOID lpvData;
    DWORD  dwStride;
} D3DDP_PTRSTRIDE;
```

Members

lpvData
Address of an array of data.

dwStride
Memory stride to between elements in the array.

Remarks

This structure can be used with a composite vertex format (like the **D3DLVERTEX** structure) or a distinct array of vertex components. In a composite vertex format, the **lpvData** points to a particular component, and the **dwStride** member is the stride, in bytes, of the composite format. In an array that contains only one vertex component, the **dwStride** member should be the stride of each element in the array.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DDRAWPRIMITIVESTRIDEDDATA, Strided Vertex Format

D3DDRAWPRIMITIVESTRIDEDDATA

[This is preliminary documentation and subject to change.]

The **D3DDRAWPRIMITIVESTRIDEDDATA** structure contains flexible vertex format components.

```
typedef struct D3DDRAWPRIMITIVESTRIDEDDATA  {
    D3DDP_PTRSTRIDE position;
    D3DDP_PTRSTRIDE normal;
    D3DDP_PTRSTRIDE diffuse;
    D3DDP_PTRSTRIDE specular;
    D3DDP_PTRSTRIDE textureCoords[D3DDP_MAXTEXCOORD];
} D3DDRAWPRIMITIVESTRIDEDDATA , *LPD3DDRAWPRIMITIVESTRIDEDDATA;
```

Members

position and **normal**
: **D3DDP_PTRSTRIDE** structures that point to arrays of position and normal vectors for a collection of vertices (each vector is a 3-element array of float values).

diffuse and **specular**
: **D3DDP_PTRSTRIDE** structures that point to diffuse and specular color information for a collection of vertices. Each color component is an 8-8-8-8 RGBA value.

textureCoords
: An 8-element array of **D3DDP_PTRSTRIDE** structures. Each element in the array is an array of texture coordinates for the collection of vertices. Your application determines which array of texture coordinates is used for a given texture stage by calling the **IDirect3DDevice3::SetTextureStageState** method with the D3DTSS_TEXCOORDINDEX stage state value.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DDP_PTRSTRIDE, Strided Vertex Format, Vertex Formats

D3DEXECUTEBUFFERDESC

[This is preliminary documentation and subject to change.]

The **D3DEXECUTEBUFFERDESC** structure describes the execute buffer for such methods as **IDirect3DDevice::CreateExecuteBuffer** and **IDirect3DExecuteBuffer::Lock**.

```
typedef struct _D3DExecuteBufferDesc {
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwCaps;
    DWORD  dwBufferSize;
    LPVOID lpData;
} D3DEXECUTEBUFFERDESC, *LPD3DEXECUTEBUFFERDESC;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Flags identifying the members of this structure that contain valid data.

D3DDEB_BUFSIZE
The **dwBufferSize** member is valid.

D3DDEB_CAPS
The **dwCaps** member is valid.

D3DDEB_LPDATA
The **lpData** member is valid.

dwCaps
Location in memory of the execute buffer.

D3DDEBCAPS_MEM
A logical **OR** of D3DDEBCAPS_SYSTEMMEMORY and D3DDEBCAPS_VIDEOMEMORY.

D3DDEBCAPS_SYSTEMMEMORY
The execute buffer data resides in system memory.

D3DDEBCAPS_VIDEOMEMORY
The execute buffer data resides in device memory.

dwBufferSize
Size of the execute buffer, in bytes.

lpData
Address of the buffer data.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dcaps.h.

D3DEXECUTEDATA

[This is preliminary documentation and subject to change.]

The **D3DEXECUTEDATA** structure specifies data for the **IDirect3DDevice::Execute** method. When this method is called and the transformation has been done, the instruction list starting at the value specified in the **dwInstructionOffset** member is parsed and rendered.

```
typedef struct _D3DEXECUTEDATA {
    DWORD     dwSize;
    DWORD     dwVertexOffset;
    DWORD     dwVertexCount;
    DWORD     dwInstructionOffset;
    DWORD     dwInstructionLength;
    DWORD     dwHVertexOffset;
    D3DSTATUS dsStatus;
} D3DEXECUTEDATA, *LPD3DEXECUTEDATA;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

dwVertexOffset
Offset into the list of vertices.

dwVertexCount
Number of vertices to execute.

dwInstructionOffset
Offset into the list of instructions to execute.

dwInstructionLength
Length of the instructions to execute.

dwHVertexOffset
Offset into the list of vertices for the homogeneous vertex used when the application is supplying screen coordinate data that needs clipping.

dsStatus
Value storing the screen extent of the rendered geometry for use after the transformation is complete. This value is a **D3DSTATUS** structure.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DSTATUS

D3DFINDDEVICERESULT

[This is preliminary documentation and subject to change.]

The **D3DFINDDEVICERESULT** structure identifies a device an application has found by calling the **IDirect3D3::FindDevice** method.

```
typedef struct _D3DFINDDEVICERESULT {
    DWORD        dwSize;
    GUID         guid;
    D3DDEVICEDESC ddHwDesc;
    D3DDEVICEDESC ddSwDesc;
} D3DFINDDEVICERESULT, *LPD3DFINDDEVICERESULT;
```

Members

dwSize
Size, in bytes, of the structure. This member must be initialized before the structure is used.

guid
Globally unique identifier (GUID) of the device that was found.

ddHwDesc and **ddSwDesc**
D3DDEVICEDESC structures describing the hardware and software devices that were found.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dcaps.h.

See Also

D3DFINDDEVICESEARCH

D3DFINDDEVICESEARCH

[This is preliminary documentation and subject to change.]

The **D3DFINDDEVICESEARCH** structure specifies the characteristics of a device an application wants to find. This structure is used in calls to the **IDirect3D3::FindDevice** method.

```
typedef struct _D3DFINDDEVICESEARCH {
    DWORD        dwSize;
    DWORD        dwFlags;
    BOOL         bHardware;
    D3DCOLORMODEL dcmColorModel;
    GUID         guid;
    DWORD        dwCaps;
    D3DPRIMCAPS   dpcPrimCaps;
} D3DFINDDEVICESEARCH, *LPD3DFINDDEVICESEARCH;
```

Members

dwSize
Size, in bytes, of this structure. This member must be initialized before the structure is used.

dwFlags
Flags defining the type of device the application wants to find. This member can be one or more of the following values:

D3DFDS_ALPHACMPCAPS
Match the **dwAlphaCmpCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_COLORMODEL
Match the color model specified in the **dcmColorModel** member of this structure.

D3DFDS_DSTBLENDCAPS
Match the **dwDestBlendCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_GUID
Match the globally unique identifier (GUID) specified in the **guid** member of this structure.

D3DFDS_HARDWARE
Match the hardware or software search specification given in the **bHardware** member of this structure.

D3DFDS_LINES
Match the **D3DPRIMCAPS** structure specified by the **dpcLineCaps** member of the **D3DDEVICEDESC** structure.

D3DFDS_MISCCAPS

Match the **dwMiscCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_RASTERCAPS
Match the **dwRasterCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_SHADECAPS
Match the **dwShadeCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_SRCBLENDCAPS
Match the **dwSrcBlendCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TEXTUREBLENDCAPS
Match the **dwTextureBlendCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TEXTURECAPS
Match the **dwTextureCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TEXTUREFILTERCAPS
Match the **dwTextureFilterCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TRIANGLES
Match the **D3DPRIMCAPS** structure specified by the **dpcTriCaps** member of the **D3DDEVICEDESC** structure.

D3DFDS_ZCMPCAPS
Match the **dwZCmpCaps** member of the **D3DPRIMCAPS** structure specified as the **dpcPrimCaps** member of this structure.

bHardware
Flag specifying whether the device to find is implemented as hardware or software. If this member is TRUE, the device to search for has hardware rasterization and may also provide other hardware acceleration. Applications that use this flag should set the D3DFDS_HARDWARE bit in the **dwFlags** member.

dcmColorModel
One of the values of the **D3DCOLORMODEL** data type, specifying whether the device to find should use the ramp or RGB color model.

guid
Globally unique identifier (GUID) of the device to find.

dwCaps
Reserved.

dpcPrimCaps
Specifies a **D3DPRIMCAPS** structure defining the device's capabilities for each primitive type.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dcaps.h.

See Also

D3DFINDDEVICERESULT

D3DHVERTEX

[This is preliminary documentation and subject to change.]

The **D3DHVERTEX** structure defines a homogeneous vertex used when the application is supplying screen coordinate data that needs clipping. This structure is part of the **D3DTRANSFORMDATA** structure.

```
typedef struct _D3DHVERTEX {
    DWORD       dwFlags;
    union {
        D3DVALUE hx;
        D3DVALUE dvHX;
    };
    union {
        D3DVALUE hy;
        D3DVALUE dvHY;
    };
    union {
        D3DVALUE hz;
        D3DVALUE dvHZ;
    };
} D3DHVERTEX, *LPD3DHVERTEX;
```

Members

dwFlags
Flags defining the clip status of the homogeneous vertex. This member can be one or more of the flags described in the **dwClip** member of the **D3DTRANSFORMDATA** structure.

dvHX, **dvHY**, and **dvHZ**
Values of the **D3DVALUE** type describing transformed homogeneous coordinates. These coordinates define the vertex.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DINSTRUCTION

[This is preliminary documentation and subject to change.]

The **D3DINSTRUCTION** structure defines an instruction in an execute buffer. A display list is made up from a list of variable length instructions. Each instruction begins with a common instruction header and is followed by the data required for that instruction.

```
typedef struct _D3DINSTRUCTION {
    BYTE bOpcode;
    BYTE bSize;
    WORD wCount;
} D3DINSTRUCTION, *LPD3DINSTRUCTION;
```

Members

bOpcode
Rendering operation, specified as a member of the **D3DOPCODE** enumerated type.

bSize
Size of each instruction data unit. This member can be used to skip to the next instruction in the sequence.

wCount
Number of data units of instructions that follow. This member allows efficient processing of large batches of similar instructions, such as triangles that make up a triangle mesh.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DLIGHT

[This is preliminary documentation and subject to change.]

For DirectX 5.0 and later, this structure is superseded by the **D3DLIGHT2** structure. The **D3DLIGHT2** structure adds a **dwFlags** member. In addition, the **dvAttenuation** members are interpreted differently in **D3DLIGHT2** than they were for **D3DLIGHT**.

```
typedef struct _D3DLIGHT {
    DWORD           dwSize;
    D3DLIGHTTYPE    dltType;
    D3DCOLORVALUE   dcvColor;
    D3DVECTOR       dvPosition;
    D3DVECTOR       dvDirection;
    D3DVALUE        dvRange;
    D3DVALUE        dvFalloff;
    D3DVALUE        dvAttenuation0;
    D3DVALUE        dvAttenuation1;
    D3DVALUE        dvAttenuation2;
    D3DVALUE        dvTheta;
    D3DVALUE        dvPhi;
} D3DLIGHT, *LPD3DLIGHT;
```

Remarks

When light properties are defined with this structure, per-vertex color (enabled through D3DLIGHTSTATE_COLORVERTEX) is not supported. You must define and set light properties in a **D3DLIGHT2** structure to use per-vertex color.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DLIGHT2

[This is preliminary documentation and subject to change.]

The **D3DLIGHT2** structure defines the light type in calls to methods such as **IDirect3DLight::SetLight** and **IDirect3DLight::GetLight**.

For DirectX 5.0 and newer, this structure supersedes the **D3DLIGHT** structure. The **D3DLIGHT2** structure is identical to **D3DLIGHT** except for the addition of the **dwFlags** member. In addition, the **dvAttenuation** members are interpreted differently in **D3DLIGHT2** than they were for **D3DLIGHT**.

```
typedef struct _D3DLIGHT2 {
    DWORD        dwSize;
    D3DLIGHTTYPE  dltType;
    D3DCOLORVALUE dcvColor;
```

```
    D3DVECTOR     dvPosition;
    D3DVECTOR     dvDirection;
    D3DVALUE      dvRange;
    D3DVALUE      dvFalloff;
    D3DVALUE      dvAttenuation0;
    D3DVALUE      dvAttenuation1;
    D3DVALUE      dvAttenuation2;
    D3DVALUE      dvTheta;
    D3DVALUE      dvPhi;
    DWORD         dwFlags;
} D3DLIGHT2, *LPD3DLIGHT2;
```

Members

dwSize
Size, in bytes, of this structure. You must specify a value for this member. Direct3D uses the specified size to determine whether this is a **D3DLIGHT** or a **D3DLIGHT2** structure.

dltType
Type of the light source. This value is one of the members of the **D3DLIGHTTYPE** enumerated type.

dcvColor
Color of the light. This member is a **D3DCOLORVALUE** structure. In ramp mode, the color is converted to a gray scale.

dvPosition
Position of the light in world space. This member has no meaning for directional lights and is ignored in that case.

dvDirection
Direction the light is pointing in world space. This member only has meaning for directional and spotlights. This vector need not be normalized but it should have a nonzero length.

dvRange
Distance beyond which the light has no effect. The maximum allowable value for this member is D3DLIGHT_RANGE_MAX, which is defined as the square root of FLT_MAX. This member does not affect directional lights.

dvFalloff
Decrease in illumination between a spotlight's inner cone (the angle specified by the **dvTheta** member) and the outer edge of the outer cone (the angle specified by the **dvPhi** member). This feature was implemented for DirectX 5.0. For details on how **dvFalloff** values affect a spotlight, see Spotlight Falloff Model.

The effect of falloff on the lighting is subtle. Furthermore, a small performance penalty is incurred by shaping the falloff curve. For these reasons, most developers set this value to 1.0.

dvAttenuation0 through **dvAttenuation2**

Values specifying how a light's intensity changes over distance. (Attenuation does not affect directional lights.) In the **D3DLIGHT2** structure these values are interpreted differently than they were for the **D3DLIGHT** structure. For information about how these attenuation values affect lighting in a scene, see Light Attenuation Over Distance.

dvTheta

Angle, in radians, of the spotlight's inner cone—that is, the fully illuminated spotlight cone. This value must be between 0 and the value specified by the **dvPhi** member.

dvPhi

Angle, in radians, defining the outer edge of the spotlight's outer cone. Points outside this cone are not lit by the spotlight. This value must be between 0 and pi.

dwFlags

A combination of the following performance-related flags. This member is new for DirectX 5.0.

D3DLIGHT_ACTIVE

Enables the light. This flag must be set to enable the light; if it is not set, the light is ignored.

D3DLIGHT_NO_SPECULAR

Turns off specular highlights for the light.

Remarks

In the **D3DLIGHT** structure, the affects of the attenuation settings were difficult to predict; developers were encouraged to experiment with the settings until they achieved the desired result. For **D3DLIGHT2**, it is much easier to work with lighting attenuation.

When you use this structure with the **IDirect3DLight::GetLight** or **IDirect3DLight::SetLight**, cast the pointer to this structure to the **LPD3DLIGHT** data type.

For more information about lights, see Lights and **IDirect3DLight**.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DLIGHTTYPE

D3DLIGHTDATA

[This is preliminary documentation and subject to change.]

The **D3DLIGHTDATA** structure describes the points to be lit and resulting colors in calls to the **IDirect3DViewport3::LightElements** method.

```
typedef struct _D3DLIGHTDATA {
    DWORD               dwSize;
    LPD3DLIGHTINGELEMENT lpIn;
    DWORD               dwInSize;
    LPD3DTLVERTEX       lpOut;
    DWORD               dwOutSize;
} D3DLIGHTDATA, *LPD3DLIGHTDATA;
```

Members

dwSize
Size, in bytes, of this structure. This member must be initialized before the structure is used.

lpIn
Address of a **D3DLIGHTINGELEMENT** structure specifying the input positions and normal vectors.

dwInSize
Amount to skip from one input element to the next. This allows the application to store extra data inline with the element.

lpOut
Address of a **D3DTLVERTEX** structure specifying the output colors.

dwOutSize
Amount to skip from one output color to the next. This allows the application to store extra data inline with the color.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DLIGHTINGCAPS

[This is preliminary documentation and subject to change.]

The **D3DLIGHTINGCAPS** structure describes the lighting capabilities of a device. This structure is a member of the **D3DDEVICEDESC** structure.

```
typedef struct _D3DLIGHTINGCAPS {
    DWORD dwSize;
    DWORD dwCaps;
    DWORD dwLightingModel;
    DWORD dwNumLights;
} D3DLIGHTINGCAPS, *LPD3DLIGHTINGCAPS;
```

Members

dwSize
Size, in bytes, of this structure. This member must be initialized before the structure is used.

dwCaps
Flags describing the capabilities of the lighting module. The following flags are defined:

D3DLIGHTCAPS_DIRECTIONAL
Supports directional lights.

D3DLIGHTCAPS_GLSPOT
Not used.

D3DLIGHTCAPS_PARALLELPOINT
Supports parallel point lights.

D3DLIGHTCAPS_POINT
Supports point lights.

D3DLIGHTCAPS_SPOT
Supports spotlights.

dwLightingModel
Flags defining whether the lighting model is RGB or monochrome. The following flags are defined:

D3DLIGHTINGMODEL_MONO
Monochromatic lighting model.

D3DLIGHTINGMODEL_RGB
RGB lighting model.

dwNumLights
Number of lights that can be handled.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dcaps.h.

D3DLIGHTINGELEMENT

[This is preliminary documentation and subject to change.]

The **D3DLIGHTINGELEMENT** structure describes the points in model space that will be lit. This structure is part of the **D3DLIGHTDATA** structure.

```
typedef struct _D3DLIGHTINGELEMENT {
    D3DVECTOR dvPosition;
    D3DVECTOR dvNormal;
} D3DLIGHTINGELEMENT, *LPD3DLIGHTINGELEMENT;
```

Members

dvPosition
Value specifying the lightable point in model space. This value is a **D3DVECTOR** structure.

dvNormal
Value specifying the normalized unit vector. This value is a **D3DVECTOR** structure.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DLIGHTDATA, **IDirect3DViewport3::LightElements**

D3DLINE

[This is preliminary documentation and subject to change.]

The **D3DLINE** structure describes a line for the D3DOP_LINE opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DLINE {
    union {
        WORD v1;
        WORD wV1;
    };
    union {
        WORD v2;
        WORD wV2;
```

```
    };
} D3DLINE, *LPD3DLINE;
```

Members

wV1 and **wV2**
Vertex indices.

Remarks

The instruction count defines the number of line segments.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DLINEPATTERN

[This is preliminary documentation and subject to change.]

The **D3DLINEPATTERN** structure describes a line pattern. These values are used by the D3DRENDERSTATE_LINEPATTERN render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef struct _D3DLINEPATTERN {
    WORD wRepeatFactor;
    WORD wLinePattern;
} D3DLINEPATTERN;
```

Members

wRepeatFactor
Number of times to duplicate each series of 1s and 0s specified in the **wLinePattern** member. This repeat factor allows an application to "stretch" the line pattern.

wLinePattern
Bits specifying the line pattern. For example, the following value would produce a dotted line: 1100110011001100.

Remarks

A line pattern specifies how a line is drawn. The line pattern is always the same, no matter where it is started. (This is as opposed to stippling, which affects how objects are rendered; that is, to imitate transparency.)

The line pattern specifies up to a 16-pixel pattern of on and off pixels along the line. The **wRepeatFactor** member specifies how many pixels are repeated for each entry in **wLinePattern**.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DLVERTEX

[This is preliminary documentation and subject to change.]

The **D3DLVERTEX** structure defines an untransformed and lit vertex (model coordinates with color). An application should use this structure when the vertex transformations will be handled by Direct3D. This structure contains only data and a color that would be filled by software lighting.

```
typedef struct _D3DLVERTEX {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
    DWORD       dwReserved;
    union {
        D3DCOLOR color;
        D3DCOLOR dcColor;
    };
    union {
        D3DCOLOR specular;
        D3DCOLOR dcSpecular;
    };
```

```
    union {
        D3DVALUE tu;
        D3DVALUE dvTU;
    };
    union {
        D3DVALUE tv;
        D3DVALUE dvTV;
    };
} D3DLVERTEX, *LPD3DLVERTEX;
```

Members

dvX, **dvY**, and **dvZ**
Values of the **D3DVALUE** type specifying the model coordinates of the vertex.

dwReserved
Reserved; must be zero.

dcColor and **dcSpecular**
Values of the **D3DCOLOR** type specifying the color and specular component of the vertex.

dvTU and **dvTV**
Values of the **D3DVALUE** type specifying the texture coordinates of the vertex.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DTLVERTEX, **D3DVERTEX**

D3DMATERIAL

[This is preliminary documentation and subject to change.]

The **D3DMATERIAL** structure specifies material properties in calls to the **IDirect3DMaterial3::GetMaterial** and **IDirect3DMaterial3::SetMaterial** methods.

```
typedef struct _D3DMATERIAL {
    DWORD           dwSize;
    union {
        D3DCOLORVALUE diffuse;
        D3DCOLORVALUE dcvDiffuse;
    };
```

```
    union {
        D3DCOLORVALUE ambient;
        D3DCOLORVALUE dcvAmbient;
    };
    union {
        D3DCOLORVALUE specular;
        D3DCOLORVALUE dcvSpecular;
    };
    union {
        D3DCOLORVALUE emissive;
        D3DCOLORVALUE dcvEmissive;
    };
    union {
        D3DVALUE      power;
        D3DVALUE      dvPower;
    };
    D3DTEXTUREHANDLE  hTexture;     // Used only by Ramp devices.
    DWORD             dwRampSize;
} D3DMATERIAL, *LPD3DMATERIAL;
```

Members

dwSize
Size, in bytes, of this structure. This member must be initialized before the structure is used.

dcvDiffuse, **dcvAmbient**, **dcvSpecular**, and **dcvEmissive**
Values specifying the diffuse color, ambient color, specular color, and emissive color of the material, respectively. These values are **D3DCOLORVALUE** structures.

dvPower
Value of the **D3DVALUE** type specifying the sharpness of specular highlights.

hTexture
Handle to the texture map for use by a ramp device. This member can be zero to indicate that the material does not use a texture or when the material is being used with a device other than the ramp device.

dwRampSize
Size of the color ramp. For the monochromatic (ramp) driver, this value should be 1 for materials assigned to the background.

Remarks

The texture handle specified by the **hTexture** member is acquired from Direct3D by loading a texture into the device. The texture handle may be used only when it has been loaded into the device. This texture handle is only used by the legacy ramp device, which is not supported by interfaces introduced in DirectX 6.0, such as the

new **IDirect3DDevice3** interface. For more information, see Ramp Device in Legacy Device Types.

To turn off specular highlights for a material, you must set the **dvPower** member to 0—simply setting the specular color components to 0 is not enough.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DMaterial3::GetMaterial, **IDirect3DMaterial3::SetMaterial**

D3DMATRIX

[This is preliminary documentation and subject to change.]

The **D3DMATRIX** structure describes a matrix for such methods as **IDirect3DDevice::GetMatrix** and **IDirect3DDevice::SetMatrix**.

C++ programmers can use an extended version of this structure that includes a parentheses ("()") operator. For more information, see **D3DMATRIX (D3D_OVERLOADS)**

```
typedef struct _D3DMATRIX {
    D3DVALUE _11, _12, _13, _14;
    D3DVALUE _21, _22, _23, _24;
    D3DVALUE _31, _32, _33, _34;
    D3DVALUE _41, _42, _43, _44;
} D3DMATRIX, *LPD3DMATRIX;
```

Remarks

In Direct3D, the _34 element of a projection matrix cannot be a negative number. If your application needs to use a negative value in this location, it should scale the entire projection matrix by -1, instead.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice::GetMatrix, **IDirect3DDevice::SetMatrix**

D3DMATRIXLOAD

[This is preliminary documentation and subject to change.]

The **D3DMATRIXLOAD** structure describes the operand data for the D3DOP_MATRIXLOAD opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DMATRIXLOAD {
    D3DMATRIXHANDLE hDestMatrix;
    D3DMATRIXHANDLE hSrcMatrix;
} D3DMATRIXLOAD, *LPD3DMATRIXLOAD;
```

Members

hDestMatrix and **hSrcMatrix**
Handles to the destination and source matrices. These values are **D3DMATRIX** structures.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DOPCODE

D3DMATRIXMULTIPLY

[This is preliminary documentation and subject to change.]

The **D3DMATRIXMULTIPLY** structure describes the operand data for the D3DOP_MATRIXMULTIPLY opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DMATRIXMULTIPLY {
    D3DMATRIXHANDLE hDestMatrix;
    D3DMATRIXHANDLE hSrcMatrix1;
    D3DMATRIXHANDLE hSrcMatrix2;
} D3DMATRIXMULTIPLY, *LPD3DMATRIXMULTIPLY;
```

Members

hDestMatrix
Handle to the matrix that stores the product of the source matrices. This value is a **D3DMATRIX** structure.

hSrcMatrix1 and **hSrcMatrix2**
Handles of the first and second source matrices. These values are **D3DMATRIX** structures.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DOPCODE

D3DPICKRECORD

[This is preliminary documentation and subject to change.]

The **D3DPICKRECORD** structure returns information about picked primitives in an execute buffer for the **IDirect3DDevice::GetPickRecords** method.

```
typedef struct _D3DPICKRECORD {
    BYTE     bOpcode;
    BYTE     bPad;
    DWORD    dwOffset;
    D3DVALUE dvZ;
} D3DPICKRECORD, *LPD3DPICKRECORD;
```

Members

bOpcode
Opcode of the picked primitive.

bPad
Pad byte.

dwOffset
Offset from the start of the instruction segment portion of the execute buffer in which the picked primitive was found. (The instruction segment portion of the execute buffer is the part of the execute buffer that follows the vertex list.)

dvZ
Depth of the picked primitive.

Remarks

The x- and y-coordinates of the picked primitive are specified in the call to the **IDirect3DDevice::Pick** method that created the pick records.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice::GetPickRecords, **IDirect3DDevice::Pick**

D3DPOINT

[This is preliminary documentation and subject to change.]

The **D3DPOINT** structure describes operand data for the D3DOP_POINT opcode in the in **D3DOPCODE** enumerated type.

```
typedef struct _D3DPOINT {
    WORD wCount;
    WORD wFirst;
} D3DPOINT, *LPD3DPOINT;
```

Members

wCount
Number of points.

wFirst
Index of the first vertex.

Remarks

Points are rendered by using a list of vertices.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DOPCODE

D3DPRIMCAPS

[This is preliminary documentation and subject to change.]

The **D3DPRIMCAPS** structure defines the capabilities for each primitive type. This structure is used when creating a device and when querying the capabilities of a device. This structure defines several members in the **D3DDEVICEDESC** structure.

```
typedef struct _D3DPrimCaps {
    DWORD dwSize;              // size of structure
    DWORD dwMiscCaps;          // miscellaneous caps
    DWORD dwRasterCaps;        // raster caps
    DWORD dwZCmpCaps;          // z-comparison caps
    DWORD dwSrcBlendCaps;      // source blending caps
    DWORD dwDestBlendCaps;     // destination blending caps
    DWORD dwAlphaCmpCaps;      // alpha-test comparison caps
    DWORD dwShadeCaps;         // shading caps
    DWORD dwTextureCaps;       // texture caps
    DWORD dwTextureFilterCaps; // texture filtering caps
    DWORD dwTextureBlendCaps;  // texture blending caps
    DWORD dwTextureAddressCaps; // texture addressing caps
    DWORD dwStippleWidth;      // stipple width
    DWORD dwStippleHeight;     // stipple height
} D3DPRIMCAPS, *LPD3DPRIMCAPS;
```

Members

dwSize
Size, in bytes, of this structure. This member must be initialized before the structure is used.

dwMiscCaps
General capabilities for this primitive. This member can be one or more of the following:

D3DPMISCCAPS_CONFORMANT
The device conforms to the OpenGL standard.

D3DPMISCCAPS_CULLCCW
The driver supports counterclockwise culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CCW member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_CULLCW

The driver supports clockwise triangle culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CW member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_CULLNONE
The driver does not perform triangle culling. This corresponds to the D3DCULL_NONE member of the **D3DCULL** enumerated type.

D3DPMISCCAPS_LINEPATTERNREP
The driver can handle values other than 1 in the **wRepeatFactor** member of the **D3DLINEPATTERN** structure. (This applies only to line-drawing primitives.)

D3DPMISCCAPS_MASKPLANES
The device can perform a bitmask of color planes.

D3DPMISCCAPS_MASKZ
The device can enable and disable modification of the depth-buffer on pixel operations.

dwRasterCaps
Information on raster-drawing capabilities. This member can be one or more of the following:

D3DPRASTERCAPS_ANISOTROPY
The device supports anisotropic filtering. For more information, see D3DRENDERSTATE_ANISOTROPY in the **D3DRENDERSTATETYPE** structure.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_ANTIALIASEDGES
The device can antialias lines forming the convex outline of objects. For more information, see D3DRENDERSTATE_EDGEANTIALIAS in the **D3DRENDERSTATETYPE** structure.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT
The device supports antialiasing that is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur. For more information, see the **D3DANTIALIASMODE** enumerated type.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT
The device supports antialiasing that is not dependent on the sort order of the polygons. For more information, see the **D3DANTIALIASMODE** enumerated type.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_DITHER
The device can dither to improve color resolution.

D3DPRASTERCAPS_FOGRANGE

The device supports range-based fog. In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene. For more information, see Range-based Fog.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_FOGTABLE
The device calculates the fog value by referring to a lookup table containing fog values that are indexed to the depth of a given pixel.

D3DPRASTERCAPS_FOGVERTEX
The device calculates the fog value during the lighting operation, places the value into the alpha component of the **D3DCOLOR** value given for the **specular** member of the **D3DTLVERTEX** structure, and interpolates the fog value during rasterization.

D3DPRASTERCAPS_MIPMAPLODBIAS
The device supports level-of-detail (LOD) bias adjustments. These bias adjustments enable an application to make a mipmap appear crisper or less sharp than it normally would. For more information about LOD bias in mipmaps, see D3DRENDERSTATE_MIPMAPLODBIAS**.**

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_PAT
The driver can perform patterned drawing (lines or fills with D3DRENDERSTATE_LINEPATTERN or one of the D3DRENDERSTATE_STIPPLEPATTERN render states) for the primitive being queried.

D3DPRASTERCAPS_ROP2
The device can support raster operations other than R2_COPYPEN.

D3DPRASTERCAPS_STIPPLE
The device can stipple polygons to simulate translucency.

D3DPRASTERCAPS_SUBPIXEL
The device performs subpixel placement of z, color, and texture data, rather than working with the nearest integer pixel coordinate. This helps avoid bleed-through due to z imprecision, and jitter of color and texture values for pixels. Note that there is no corresponding state that can be enabled and disabled; the device either performs subpixel placement or it does not, and this bit is present only so that the Direct3D client will be better able to determine what the rendering quality will be.

D3DPRASTERCAPS_SUBPIXELX
The device is subpixel accurate along the x-axis only and is clamped to an integer y-axis scan line. For information about subpixel accuracy, see D3DPRASTERCAPS_SUBPIXEL.

D3DPRASTERCAPS_TRANSLUCENTSORTINDEPENDENT
The device supports translucency that is not dependent on the sort order of the polygons. For more information, see the D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT.

D3DPRASTERCAPS_WBUFFER
The device supports depth buffering using w. For more information, see Depth Buffers.

D3DPRASTERCAPS_WFOG
The device supports w-based fog. W-based fog is used when a perspective projection matrix is specified, but affine projections will still use z-based fog. The system considers a projection matrix that contains a non-zero value in the [3][4] element to be a perspective projection matrix.

D3DPRASTERCAPS_XOR
The device can support **XOR** operations. If this flag is not set but D3DPRIM_RASTER_ROP2 is set, then **XOR** operations must still be supported.

D3DPRASTERCAPS_ZBIAS
The device supports z-bias values. These are integer values assigned to polygons that allow physically coplanar polygons to appear separate. For more information, see D3DRENDERSTATE_ZBIAS in the **D3DRENDERSTATETYPE** enumerated type.

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_ZBUFFERLESSHSR
The device can perform hidden-surface removal (HSR) without requiring the application to sort polygons, and without requiring the allocation of a depth-buffer. This leaves more video memory for textures. The method used to perform hidden-surface removal is hardware-dependent and is transparent to the application.

Z-bufferless HSR is performed if no depth-buffer surface is attached to the rendering-target surface and the depth-buffer comparison test is enabled (that is, when the state value associated with the D3DRENDERSTATE_ZENABLE enumeration constant is set to TRUE).

This flag was introduced in DirectX 5.0.

D3DPRASTERCAPS_ZTEST
The device can perform z-test operations. This effectively renders a primitive and indicates whether any z pixels would have been rendered.

dwZCmpCaps
Z-buffer comparison functions that the driver can perform. This member can be one or more of the following:

D3DPCMPCAPS_ALWAYS
Always pass the z test.

D3DPCMPCAPS_EQUAL
Pass the z test if the new z equals the current z.

D3DPCMPCAPS_GREATER
Pass the z test if the new z is greater than the current z.

D3DPCMPCAPS_GREATEREQUAL
Pass the z test if the new z is greater than or equal to the current z.

D3DPCMPCAPS_LESS

Pass the z test if the new z is less than the current z.

D3DPCMPCAPS_LESSEQUAL
Pass the z test if the new z is less than or equal to the current z.

D3DPCMPCAPS_NEVER
Always fail the z test.

D3DPCMPCAPS_NOTEQUAL
Pass the z test if the new z does not equal the current z.

dwSrcBlendCaps
Source blending capabilities. This member can be one or more of the following. (The RGBA values of the source and destination are indicated with the subscripts *s* and *d*.)

D3DPBLENDCAPS_BOTHINVSRCALPHA
Source blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$) and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden.

D3DPBLENDCAPS_BOTHSRCALPHA
The driver supports the D3DBLEND_BOTHSRCALPHA blend mode. (This blend mode is obsolete for DirectX 6.0 and later. For more information, see **D3DBLEND**.)

D3DPBLENDCAPS_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DPBLENDCAPS_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DPBLENDCAPS_INVDESTALPHA
Blend factor is ($1-A_d$, $1-A_d$, $1-A_d$, $1-A_d$).

D3DPBLENDCAPS_INVDESTCOLOR
Blend factor is ($1-R_d$, $1-G_d$, $1-B_d$, $1-A_d$).

D3DPBLENDCAPS_INVSRCALPHA
Blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$).

D3DPBLENDCAPS_INVSRCCOLOR
Blend factor is ($1-R_d$, $1-G_d$, $1-B_d$, $1-A_d$).

D3DPBLENDCAPS_ONE
Blend factor is (1, 1, 1, 1).

D3DPBLENDCAPS_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DPBLENDCAPS_SRCALPHASAT
Blend factor is (f, f, f, 1); f = min(A_s, $1-A_d$).

D3DPBLENDCAPS_SRCCOLOR
Blend factor is (R_s, G_s, B_s, A_s).

D3DPBLENDCAPS_ZERO
Blend factor is (0, 0, 0, 0).

dwDestBlendCaps
Destination blending capabilities. This member can be the same capabilities that are defined for the **dwSrcBlendCaps** member.

dwAlphaCmpCaps

Alpha-test comparison functions that the driver can perform. This member can include the same capability flags defined for the **dwZCmpCaps** member. If this member contains only the D3DPCMPCAPS_ALWAYS capability or only the D3DPCMPCAPS_NEVER capability, the driver does not support alpha tests. Otherwise, the flags identify the individual comparisons that are supported for alpha testing.

dwShadeCaps

Shading operations that the device can perform. It is assumed, in general, that if a device supports a given command (such as D3DOP_TRIANGLE) at all, it supports the D3DSHADE_FLAT mode (as specified in the **D3DSHADEMODE** enumerated type). This flag specifies whether the driver can also support Gouraud and Phong shading and whether alpha color components are supported for each of the three color-generation modes. When alpha components are not supported in a given mode, the alpha value of colors generated in that mode is implicitly 255. This is the maximum possible alpha (that is, the alpha component is at full intensity).

With the monochromatic shade modes, the blue channel of the specular component is interpreted as a white intensity. (This is controlled by the D3DRENDERSTATE_MONOENABLE render state.)

The color, specular highlights, fog, and alpha interpolants of a triangle each have capability flags that an application can use to find out how they are implemented by the device driver. These are modified by the shade mode, color model, and by whether the alpha component of a color is blended or stippled. For more information, see 3-D Primitives.

This member can be one or more of the following:

D3DPSHADECAPS_ALPHAFLATBLEND
D3DPSHADECAPS_ALPHAFLATSTIPPLED
Device can support an alpha component for flat blended and stippled transparency, respectively (the D3DSHADE_FLAT state for the **D3DSHADEMODE** enumerated type). In these modes, the alpha color component for a primitive is provided as part of the color for the first vertex of the primitive.

D3DPSHADECAPS_ALPHAGOURAUDBLEND
D3DPSHADECAPS_ALPHAGOURAUDSTIPPLED
Device can support an alpha component for Gouraud blended and stippled transparency, respectively (the D3DSHADE_GOURAUD state for the **D3DSHADEMODE** enumerated type). In these modes, the alpha color component for a primitive is provided at vertices and interpolated across a face along with the other color components.

D3DPSHADECAPS_ALPHAPHONGBLEND
D3DPSHADECAPS_ALPHAPHONGSTIPPLED
Device can support an alpha component for Phong blended and stippled transparency, respectively (the D3DSHADE_PHONG state for the **D3DSHADEMODE** enumerated type). In these modes, vertex parameters are

reevaluated on a per-pixel basis, applying lighting effects for the red, green, and blue color components. Phong shading is not currently supported.

D3DPSHADECAPS_COLORFLATMONO
D3DPSHADECAPS_COLORFLATRGB
Device can support colored flat shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, the color component for a primitive is provided as part of the color for the first vertex of the primitive. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORGOURAUDMONO
D3DPSHADECAPS_COLORGOURAUDRGB
Device can support colored Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, the color component for a primitive is provided at vertices and interpolated across a face along with the other color components. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORPHONGMONO
D3DPSHADECAPS_COLORPHONGRGB
Device can support colored Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, vertex parameters are reevaluated on a per-pixel basis. Lighting effects are applied for the red, green, and blue color components in RGB mode, and for the blue component only for monochromatic mode. Phong shading is not currently supported.

D3DPSHADECAPS_FOGFLAT
D3DPSHADECAPS_FOGGOURAUD
D3DPSHADECAPS_FOGPHONG
Device can support fog in the flat, Gouraud, and Phong shading models, respectively. Phong shading is not currently supported.

D3DPSHADECAPS_SPECULARFLATMONO
D3DPSHADECAPS_SPECULARFLATRGB
Device can support specular highlights in flat shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARGOURAUDMONO
D3DPSHADECAPS_SPECULARGOURAUDRGB
Device can support specular highlights in Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARPHONGMONO
D3DPSHADECAPS_SPECULARPHONGRGB
Device can support specular highlights in Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. Phong shading is not currently supported.

dwTextureCaps

Miscellaneous texture-mapping capabilities. This member can be one or more of the following:

D3DPTEXTURECAPS_ALPHA
Supports RGBA textures in the D3DTBLEND_DECAL and D3DTBLEND_MODULATE texture filtering modes. If this capability is not set, then only RGB textures are supported in those modes. Regardless of the setting of this flag, alpha must always be supported in D3DTBLEND_DECALMASK, D3DTBLEND_DECALALPHA, and D3DTBLEND_MODULATEALPHA filtering modes whenever those filtering modes are available.

D3DPTEXTURECAPS_ALPHAPALETTE
Supports palettized texture surfaces whose palettes contain alpha information (see DDPCAPS_ALPHA in the **DDCAPS** structure).

D3DPTEXTURECAPS_BORDER
Superseded by D3DPTADDRESSCAPS_BORDER.

D3DPTEXTURECAPS_NONPOW2CONDITIONAL
Conditionally supports the use of textures with dimensions that are not powers of two. A device that exposes this capability can use such a texture if all of the following requirements are met.

- The texture addressing mode for the texture stage is set to D3DTADDRESS_CLAMP.
- Texture wrapping for the texture stage is disabled (D3DRENDERSTATE_WRAP*n* set to zero).
- Mipmapping is not in use. (Mipmapped textures must have dimensions that are powers of two.)
- Anisotropic texture filtering is disabled.

D3DPTEXTURECAPS_PERSPECTIVE
Perspective correction is supported.

D3DPTEXTURECAPS_POW2
All nonmipmapped textures must have widths and heights specified as powers of two if this flag is set. (Note that all mipmapped textures must always have dimensions that are powers of two.)

D3DPTEXTURECAPS_SQUAREONLY
All textures must be square.

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE
Texture indices are not scaled by the texture size prior to interpolation.

D3DPTEXTURECAPS_TRANSPARENCY
Texture transparency is supported. (Only those texels that are not the current transparent color are drawn.)

dwTextureFilterCaps
Texture-map filtering capabilities. General texture filtering flags reflect which texture filtering modes you can set for the D3DRENDERSTATE_TEXTUREMAG, D3DRENDERSTATE_TEXTUREMIN render states. Per-stage filtering

capabilities reflect which filtering modes are supported for texture stages when performing multiple texture blending with the **IDirect3DDevice3** interface. This member be can any combination of the following general and per-stage texture filtering flags:

General texture filtering flags

D3DPTFILTERCAPS_LINEAR
Bilinear filtering. Chooses the texel that has nearest coordinates, then performs a weighted average with the four surrounding texels to determine the final color. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DPTFILTERCAPS_LINEARMIPLINEAR
Trilinear interpolation between mipmaps. Performs bilinear filtering on the two nearest mipmaps, then interpolates linearly between the two colors to determine a final color.

D3DPTFILTERCAPS_LINEARMIPNEAREST
Linear interpolation between two point sampled mipmaps. Chooses the nearest texel from the two closest mipmap levels, then performs linear interpolation between them.

D3DPTFILTERCAPS_MIPLINEAR
Nearest mipmapping, with bilinear filtering applied to the result. Chooses the texel from the appropriate mipmap that has nearest coordinates, then performs a weighted average with the four surrounding texels to determine the final color.

D3DPTFILTERCAPS_MIPNEAREST
Nearest mipmapping. Chooses the texel from the appropriate mipmap with coordinates nearest to the desired pixel value.

D3DPTFILTERCAPS_NEAREST
Point sampling. The texel with coordinates nearest to the desired pixel value is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

Per-stage texture filtering flags

D3DPTFILTERCAPS_MAGFAFLATCUBIC
The device supports per-stage flat-cubic filtering for magnifying textures. The flat-cubic magnification filter is represented by the D3DTFG_FLATCUBIC member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MAGFANISOTROPIC
The device supports per-stage anisotropic filtering for magnifying textures. The anisotropic magnification filter is represented by the D3DTFG_ANISOTROPIC member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC
The device supports the per-stage Gaussian-cubic filtering for magnifying textures. The Gaussian-cubic magnification filter is represented by the D3DTFG_GAUSSIANCUBIC member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MAGFLINEAR
The device supports per-stage bilinear-interpolation filtering for magnifying textures. The bilinear-interpolation magnification filter is represented by the D3DTFG_LINEAR member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MAGFPOINT
The device supports per-stage point-sampled filtering for magnifying textures. The point-sample magnification filter is represented by the D3DTFG_POINT member of the **D3DTEXTUREMAGFILTER** enumerated type.

D3DPTFILTERCAPS_MINFANISOTROPIC
The device supports per-stage anisotropic filtering for minifying textures. The anisotropic minification filter is represented by the D3DTFN_ANISOTROPIC member of the **D3DTEXTUREMINFILTER** enumerated type.

D3DPTFILTERCAPS_MINFLINEAR
The device supports per-stage bilinear-interpolation filtering for minifying textures. The bilinear minification filter is represented by the D3DTFN_LINEAR member of the **D3DTEXTUREMINFILTER** enumerated type.

D3DPTFILTERCAPS_MINFPOINT
The device supports per-stage point-sampled filtering for minifying textures. The point-sample minification filter is represented by the D3DTFN_POINT member of the **D3DTEXTUREMINFILTER** enumerated type.

D3DPTFILTERCAPS_MIPFLINEAR
The device supports per-stage trilinear-interpolation filtering for mipmaps. The trilinear-interpolation mipmapping filter is represented by the D3DTFP_LINEAR member of the **D3DTEXTUREMIPFILTER** enumerated type.

D3DPTFILTERCAPS_MIPFPOINT
The device supports per-stage point-sampled filtering for mipmaps. The point-sample mipmapping filter is represented by the D3DTFP_POINT member of the **D3DTEXTUREMIPFILTER** enumerated type.

dwTextureBlendCaps
Texture-blending capabilities. See the **D3DTEXTUREBLEND** enumerated type for discussions of the various texture-blending modes. This member can be one or more of the following:

D3DPTBLENDCAPS_ADD
Supports the additive texture-blending mode, in which the Gouraud interpolants are added to the texture lookup with saturation semantics. This capability corresponds to the D3DTBLEND_ADD member of the **D3DTEXTUREBLEND** enumerated type.

This flag was introduced in DirectX 5.0.

D3DPTBLENDCAPS_COPY
Copy mode texture-blending (D3DTBLEND_COPY from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_DECAL

Decal texture-blending mode (D3DTBLEND_DECAL from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_DECALALPHA
Decal-alpha texture-blending mode (D3DTBLEND_DECALALPHA from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_DECALMASK
Decal-mask texture-blending mode (D3DTBLEND_DECALMASK from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_MODULATE
Modulate texture-blending mode (D3DTBLEND_MODULATE from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_MODULATEALPHA
Modulate-alpha texture-blending mode (D3DTBLEND_MODULATEALPHA from the **D3DTEXTUREBLEND** enumerated type) is supported.

D3DPTBLENDCAPS_MODULATEMASK
Modulate-mask texture-blending mode (D3DTBLEND_MODULATEMASK from the **D3DTEXTUREBLEND** enumerated type) is supported.

dwTextureAddressCaps
Texture-addressing capabilities. This member can be one or more of the following:

D3DPTADDRESSCAPS_BORDER
Device supports setting coordinates outside the range [0.0, 1.0] to the border color, as specified by the D3DRENDERSTATE_BORDERCOLOR render state. This ability corresponds to the **D3DTADDRESS_BORDER** texture-addressing mode.

This flag was introduced in DirectX 5.0.

D3DPTADDRESSCAPS_CLAMP
Device can clamp textures to addresses.

D3DPTADDRESSCAPS_INDEPENDENTUV
Device can separate the texture-addressing modes of the u and v coordinates of the texture. This ability corresponds to the D3DRENDERSTATE_TEXTUREADDRESSU and D3DRENDERSTATE_TEXTUREADDRESSV render-state values.

This flag was introduced in DirectX 5.0.

D3DPTADDRESSCAPS_MIRROR
Device can mirror textures to addresses.

D3DPTADDRESSCAPS_WRAP
Device can wrap textures to addresses.

dwStippleWidth and **dwStippleHeight**
Maximum width and height of the supported stipple (up to 32×32).

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dcaps.h.

D3DPROCESSVERTICES

[This is preliminary documentation and subject to change.]

The **D3DPROCESSVERTICES** structure describes how vertices in the execute buffer should be handled by the driver. This is used by the D3DOP_PROCESSVERTICES opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DPROCESSVERTICES {
    DWORD dwFlags;
    WORD  wStart;
    WORD  wDest;
    DWORD dwCount;
    DWORD dwReserved;
} D3DPROCESSVERTICES, *LPD3DPROCESSVERTICES;
```

Members

dwFlags
One or more of the following flags indicating how the driver should process the vertices:

D3DPROCESSVERTICES_COPY
Vertices should simply be copied to the driver, because they have always been transformed and lit. If all the vertices in the execute buffer can be copied, the driver does not need to do the work of processing the vertices, and a performance improvement results.

D3DPROCESSVERTICES_NOCOLOR
Vertices should not be colored.

D3DPROCESSVERTICES_OPMASK
Specifies a bitmask of the other flags in the **dwFlags** member, exclusive of D3DPROCESSVERTICES_NOCOLOR and D3DPROCESSVERTICES_UPDATEEXTENTS.

D3DPROCESSVERTICES_TRANSFORM
Vertices should be transformed.

D3DPROCESSVERTICES_TRANSFORMLIGHT
Vertices should be transformed and lit.

D3DPROCESSVERTICES_UPDATEEXTENTS
Extents of all transformed vertices should be updated. This information is returned in the **drExtent** member of the **D3DSTATUS** structure.

wStart
Index of the first vertex in the source.

wDest
Index of the first vertex in the local buffer.

dwCount
Number of vertices to be processed.

dwReserved
Reserved; must be zero.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DOPCODE

D3DRECT

[This is preliminary documentation and subject to change.]

The **D3DRECT** structure is a rectangle definition.

```
typedef struct _D3DRECT {
    union {
        LONG x1;
        LONG lX1;
    };
    union {
        LONG y1;
        LONG lY1;
    };
    union {
        LONG x2;
        LONG lX2;
    };
    union {
        LONG y2;
        LONG lY2;
    };
} D3DRECT, *LPD3DRECT;
```

Members

lX1 and **lY1**
Coordinates of the upper-left corner of the rectangle.

lX2 and **lY2**
Coordinates of the lower-right corner of the rectangle.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice::Pick, **IDirect3DViewport3::Clear**

D3DSPAN

[This is preliminary documentation and subject to change.]

The **D3DSPAN** structure defines a span for the D3DOP_SPAN opcode in the **D3DOPCODE** enumerated type. Spans join a list of points with the same y-value. If the y-value changes, a new span is started.

```
typedef struct _D3DSPAN {
    WORD wCount;
    WORD wFirst;
} D3DSPAN, *LPD3DSPAN;
```

Members

wCount
Number of spans.

wFirst
Index to first vertex.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DOPCODE

D3DSTATE

[This is preliminary documentation and subject to change.]

The **D3DSTATE** structure describes the render state for the D3DOP_STATETRANSFORM, D3DOP_STATELIGHT, and D3DOP_STATERENDER opcodes in the **D3DOPCODE** enumerated type. The first member of this structure is the relevant enumerated type and the second is the value for that type.

```
typedef struct _D3DSTATE {
    union {
        D3DTRANSFORMSTATETYPE dtstTransformStateType;
        D3DLIGHTSTATETYPE     dlstLightStateType;
        D3DRENDERSTATETYPE    drstRenderStateType;
    };
    union {
        DWORD                 dwArg[1];
        D3DVALUE              dvArg[1];
    };
} D3DSTATE, *LPD3DSTATE;
```

Members

dtstTransformStateType, **dlstLightStateType**, and **drstRenderStateType**
One of the members of the **D3DTRANSFORMSTATETYPE**, **D3DLIGHTSTATETYPE**, or **D3DRENDERSTATETYPE** enumerated type specifying the render state.

dvArg
Value of the type specified in the first member of this structure.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DLIGHTSTATETYPE, **D3DOPCODE**, **D3DRENDERSTATETYPE**, and **D3DTRANSFORMSTATETYPE**, **D3DVALUE**

D3DSTATS

[This is preliminary documentation and subject to change.]

The **D3DSTATS** structure contains statistics used by the **IDirect3DDevice3::GetStats** method.

```
typedef struct _D3DSTATS {
    DWORD dwSize;
    DWORD dwTrianglesDrawn;
    DWORD dwLinesDrawn;
    DWORD dwPointsDrawn;
    DWORD dwSpansDrawn;
    DWORD dwVerticesProcessed;
} D3DSTATS, *LPD3DSTATS;
```

Members

dwSize
Size, in bytes, of this structure. This member must be initialized before the structure is used.

dwTrianglesDrawn, **dwLinesDrawn**, **dwPointsDrawn**, and **dwSpansDrawn**
Number of triangles, lines, points, and spans drawn since the device was created.

dwVerticesProcessed
Number of vertices processed since the device was created.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice3::GetStats

D3DSTATUS

[This is preliminary documentation and subject to change.]

The **D3DSTATUS** structure describes the current status of the execute buffer. This structure is part of the **D3DEXECUTEDATA** structure and is used with the D3DOP_SETSTATUS opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DSTATUS {
    DWORD   dwFlags;
```

```
    DWORD   dwStatus;
    D3DRECT drExtent;
} D3DSTATUS, *LPD3DSTATUS;
```

Members

dwFlags

One of the following flags, specifying whether the status, the extents, or both are being set:

D3DSETSTATUS_STATUS
Set the status.

D3DSETSTATUS_EXTENTS
Set the extents specified in the **drExtent** member.

D3DSETSTATUS_ALL
Set both the status and the extents.

dwStatus

Clipping flags. This member can be one or more of the following flags:

Combination and General Flags

D3DSTATUS_CLIPINTERSECTIONALL
Combination of all CLIPINTERSECTION flags.

D3DSTATUS_CLIPUNIONALL
Combination of all CLIPUNION flags.

D3DSTATUS_DEFAULT
Combination of D3DSTATUS_CLIPINTERSECTIONALL and D3DSTATUS_ZNOTVISIBLE flags. This value is the default.

D3DSTATUS_ZNOTVISIBLE
Indicates that the rendered primitive is not visible. This flag is set or cleared by the system when rendering with z-checking enabled (see D3DRENDERSTATE_ZVISIBLE).

Clip Intersection Flags

D3DSTATUS_CLIPINTERSECTIONBACK
Logical **AND** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM
Logical **AND** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT
Logical **AND** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through
D3DSTATUS_CLIPINTERSECTIONGEN5
Logical **AND** of the clip flags for application-defined clipping planes.

D3DSTATUS_CLIPINTERSECTIONLEFT

Logical **AND** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT
Logical **AND** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP
Logical **AND** of the clip flags for the vertices compared to the top of the viewing frustum.

Clip Union Flags

D3DSTATUS_CLIPUNIONBACK
Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM
Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT
Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5
Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT
Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT
Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP
Equal to D3DCLIP_TOP.

Basic Clipping Flags

D3DCLIP_BACK
All vertices are clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM
All vertices are clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT
All vertices are clipped by the front plane of the viewing frustum.

D3DCLIP_LEFT
All vertices are clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT
All vertices are clipped by the right plane of the viewing frustum.

D3DCLIP_TOP
All vertices are clipped by the top plane of the viewing frustum.

D3DCLIP_GEN0 through D3DCLIP_GEN5
Application-defined clipping planes.

drExtent
A **D3DRECT** structure that defines a bounding box for all the relevant vertices. For example, the structure might define the area containing the output of the D3DOP_PROCESSVERTICES opcode, assuming the

D3DPROCESSVERTICES_UPDATEEXTENTS flag is set in the **D3DPROCESSVERTICES** structure.

Remarks

The status is a rolling status and is updated during each execution. The bounding box in the **drExtent** member can grow with each execution, but it does not shrink; it can be reset only by using the D3DOP_SETSTATUS opcode.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DEXECUTEDATA, **D3DOPCODE**, **D3DRECT**

D3DTEXTURELOAD

[This is preliminary documentation and subject to change.]

The **D3DTEXTURELOAD** structure describes operand data for the D3DOP_TEXTURELOAD opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DTEXTURELOAD {
    D3DTEXTUREHANDLE hDestTexture;
    D3DTEXTUREHANDLE hSrcTexture;
} D3DTEXTURELOAD, *LPD3DTEXTURELOAD;
```

Members

hDestTexture
Handle to the destination texture.

hSrcTexture
Handle to the source texture.

Remarks

The textures referred to by the **hDestTexture** and **hSrcTexture** members must be the same size.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DTLVERTEX

[This is preliminary documentation and subject to change.]

The **D3DTLVERTEX** structure defines a transformed and lit vertex (screen coordinates with color) for the **D3DLIGHTDATA** structure.

```
typedef struct _D3DTLVERTEX {
    union {
        D3DVALUE sx;
        D3DVALUE dvSX;
    };
    union {
        D3DVALUE sy;
        D3DVALUE dvSY;
    };
    union {
        D3DVALUE sz;
        D3DVALUE dvSZ;
    };
    union {
        D3DVALUE rhw;
        D3DVALUE dvRHW;
    };
    union {
        D3DCOLOR color;
        D3DCOLOR dcColor;
    };
    union {
        D3DCOLOR specular;
        D3DCOLOR dcSpecular;
    };
    union {
        D3DVALUE tu;
        D3DVALUE dvTU;
    };
    union {
        D3DVALUE tv;
        D3DVALUE dvTV;
    };
```

```
} D3DTLVERTEX, *LPD3DTLVERTEX;
```

Members

dvSX, **dvSY**, and **dvSZ**
Values of the **D3DVALUE** type describing a vertex in screen coordinates. The largest allowable value for **dvSZ** is 0.99999, if you want the vertex to be within the range of z-values that are displayed.

dvRHW
Value of the **D3DVALUE** type that is the reciprocal of homogeneous w from homogeneous coordinate (x,y,z,w). This value is often 1 divided by the distance from the origin to the object along the z-axis.

dcColor and **dcSpecular**
Values of the **D3DCOLOR** type describing the color and specular component of the vertex.

dvTU and **dvTV**
Values of the **D3DVALUE** type describing the texture coordinates of the vertex.

Remarks

Direct3D uses the current viewport parameters (the **dwX**, **dwY**, **dwWidth**, and **dwHeight** members of the **D3DVIEWPORT2** structure) to clip **D3DTLVERTEX** vertices. The system always clips z-coordinates to [0, 1]. To prevent the system from clipping these vertices, use the D3DDP_DONOTCLIP flag in your call to **IDirect3DDevice3::Begin**.

Prior to DirectX 5.0, Direct3D did not clip **D3DTLVERTEX** vertices.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DLIGHTDATA, **D3DLVERTEX**, **D3DVERTEX**

D3DTRANSFORMCAPS

[This is preliminary documentation and subject to change.]

The **D3DTRANSFORMCAPS** structure describes the transformation capabilities of a device. This structure is part of the **D3DDEVICEDESC** structure.

```
typedef struct _D3DTransformCaps {
```

```
    DWORD dwSize;
    DWORD dwCaps;
} D3DTRANSFORMCAPS, *LPD3DTRANSFORMCAPS;
```

Members

dwSize

Size, in bytes, of this structure. This member must be initialized before the structure is used.

dwCaps

Flag specifying whether the system clips while transforming. This member can be zero or the following flag:

D3DTRANSFORMCAPS_CLIP

The system clips while transforming.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dcaps.h.

D3DTRANSFORMDATA

[This is preliminary documentation and subject to change.]

The **D3DTRANSFORMDATA** structure contains information about transformations for the **IDirect3DViewport3::TransformVertices** method.

```
typedef struct _D3DTRANSFORMDATA {
    DWORD       dwSize;
    LPVOID      lpIn;
    DWORD       dwInSize;
    LPVOID      lpOut;
    DWORD       dwOutSize;
    LPD3DHVERTEX lpHOut;
    DWORD       dwClip;
    DWORD       dwClipIntersection;
    DWORD       dwClipUnion;
    D3DRECT     drExtent;
} D3DTRANSFORMDATA, *LPD3DTRANSFORMDATA;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

lpIn
Address of the vertices to be transformed. This should be a **D3DLVERTEX** structure.

dwInSize
Stride of the vertices to be transformed.

lpOut
Address used to store the transformed vertices.

dwOutSize
Stride of output vertices.

lpHOut
Address of a value that contains homogeneous transformed vertices. This value is a **D3DHVERTEX** structure

dwClip
Flags specifying how the vertices are clipped. This member can be one or more of the following values:

D3DCLIP_BACK
Clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM
Clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT
Clipped by the front plane of the viewing frustum.

D3DCLIP_GEN0 through D3DCLIP_GEN5
Application-defined clipping planes.

D3DCLIP_LEFT
Clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT
Clipped by the right plane of the viewing frustum.

D3DCLIP_TOP
Clipped by the top plane of the viewing frustum_dx_viewing_frustum_glos.

dwClipIntersection
Flags denoting the intersection of the clip flags. This member can be one or more of the following values:

D3DSTATUS_CLIPINTERSECTIONBACK
Logical **AND** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM
Logical **AND** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT
Logical **AND** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through D3DSTATUS_CLIPINTERSECTIONGEN5
Logical **AND** of the clip flags for application-defined clipping planes.

D3DSTATUS_CLIPINTERSECTIONLEFT
Logical **AND** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT
Logical **AND** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP
Logical **AND** of the clip flags for the vertices compared to the top of the viewing frustum.

dwClipUnion
Flags denoting the union of the clip flags. This member can be one or more of the following values:

D3DSTATUS_CLIPUNIONBACK
Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM
Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT
Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5
Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT
Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT
Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP
Equal to D3DCLIP_TOP.

drExtent
A **D3DRECT** structure that defines the extents of the transformed vertices. Initialize this structure to initial extents that the **IDirect3DViewport3::TransformVertices** method will adjust if the transformed vertices do not fit. For geometries that are clipped, extents will only include vertices that are inside the viewing volume.

Remarks

Each input vertex should be a three-vector vertex giving the [x y z] coordinates in model space for the geometry. The **dwInSize** member gives the amount to skip between vertices, allowing the application to store extra data inline with each vertex.

All values generated by the transformation module are stored as 16-bit precision values. The clip is treated as an integer bitfield that is set to the inclusive **OR** of the viewing volume planes that clip a given transformed vertex.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DViewport3::TransformVertices

D3DTRIANGLE

[This is preliminary documentation and subject to change.]

The **D3DTRIANGLE** structure describes the base type for all triangles. The triangle is the main rendering primitive.

For related information, see the D3DOP_TRIANGLE member in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DTRIANGLE {
    union {
        WORD v1;
        WORD wV1;
    };
    union {
        WORD v2;
        WORD wV2;
    };
    union {
        WORD v3;
        WORD wV3;
    };
    WORD     wFlags;
} D3DTRIANGLE, *LPD3DTRIANGLE;
```

Members

wV1, **wV2**, and **wV3**
Vertices describing the triangle.

wFlags
This value can be a combination of the following flags:

Edge flags

These flags describe which edges of the triangle to enable. (This information is useful only in wireframe mode.)

D3DTRIFLAG_EDGEENABLE1
Edge defined by **v1**–**v2**.

D3DTRIFLAG_EDGEENABLE2
Edge defined by **v2**–**v3**.

D3DTRIFLAG_EDGEENABLE3
Edge defined by **v3**–**v1**.

D3DTRIFLAG_EDGEENABLETRIANGLE
All edges.

Strip and fan flags

D3DTRIFLAG_EVEN
The **v1**–**v2** edge of the current triangle is adjacent to the **v3**–**v1** edge of the previous triangle; that is, **v1** is the previous **v1**, and **v2** is the previous **v3**.

D3DTRIFLAG_ODD
The **v1**–**v2** edge of the current triangle is adjacent to the **v2**–**v3** edge of the previous triangle; that is, **v1** is the previous **v3**, and **v2** is the previous **v2**.

D3DTRIFLAG_START
Begin the strip or fan, loading all three vertices.

D3DTRIFLAG_STARTFLAT(len)
Cull or render the triangles in the strip or fan based on the treatment of this triangle. That is, if this triangle is culled, also cull the specified number of subsequent triangles. If this triangle is rendered, also render the specified number of subsequent triangles.

This length must be greater than zero and less than 30.

Remarks

This structure can be used directly for all triangle fills. For flat shading, the color and specular components are taken from the first vertex. The three vertex indices **v1**, **v2**, and **v3** are vertex indexes into the vertex list at the start of the execute buffer.

Enabled edges are visible in wireframe mode. When an application displays wireframe triangles that share an edge, it typically enables only one (or neither) edge to avoid drawing the edge twice.

The D3DTRIFLAG_ODD and D3DTRIFLAG_EVEN flags refer to the locations of a triangle in a conventional triangle strip or fan. If a triangle strip had five triangles, the following flags would be used to define the strip:

D3DTRIFLAG_START
D3DTRIFLAG_ODD
D3DTRIFLAG_EVEN
D3DTRIFLAG_ODD
D3DTRIFLAG_EVEN

Similarly, the following flags would define a triangle fan with five triangles:

D3DTRIFLAG_START
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN

The following flags could define a flat triangle fan with five triangles:

D3DTRIFLAG_STARTFLAT(4)
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN
D3DTRIFLAG_EVEN

For more information, see Triangle Strips and Triangle Fans.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DVECTOR

[This is preliminary documentation and subject to change.]

The **D3DVECTOR** structure defines a vector for many Direct3D and Direct3DRM methods and structures.

```
typedef struct _D3DVECTOR {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
} D3DVECTOR, *LPD3DVECTOR;
```

Members

dvX, **dvY**, and **dvZ**
Values of the **D3DVALUE** type describing the vector.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DLIGHT2, **D3DLIGHTINGELEMENT**

D3DVERTEX

[This is preliminary documentation and subject to change.]

The **D3DVERTEX** structure defines an untransformed and unlit vertex (model coordinates with normal direction vector).

For related information, see the D3DOP_TRIANGLE member in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DVERTEX {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
    union {
        D3DVALUE nx;
        D3DVALUE dvNX;
    };
    union {
        D3DVALUE ny;
        D3DVALUE dvNY;
    };
```

```
    union {
        D3DVALUE nz;
        D3DVALUE dvNZ;
    };
    union {
        D3DVALUE tu;
        D3DVALUE dvTU;
    };
    union {
        D3DVALUE tv;
        D3DVALUE dvTV;
    };
} D3DVERTEX, *LPD3DVERTEX;
```

Members

dvX, **dvY**, and **dvZ**
Values of the **D3DVALUE** type describing the homogeneous coordinates of the vertex.

dvNX, **dvNY**, and **dvNZ**
Values of the **D3DVALUE** type describing the normal coordinates of the vertex.

dvTU and **dvTV**
Values of the **D3DVALUE** type describing the texture coordinates of the vertex.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DLVERTEX, **D3DTLVERTEX**, **D3DVALUE**

D3DVERTEXBUFFERDESC

[This is preliminary documentation and subject to change.]

The **D3DVERTEXBUFFERDESC** structure describes the properties of a vertex buffer object. This structure is used with the **IDirect3D3::CreateVertexBuffer** and **IDirect3DVertexBuffer::GetVertexBufferDesc** methods.

```
typedef struct _D3DVERTEXBUFFERDESC {
    DWORD dwSize;
    DWORD dwCaps;
```

```
    DWORD dwFVF;
    DWORD dwNumVertices;
} D3DVERTEXBUFFERDESC, *LPD3DVERTEXBUFFERDESC;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

dwCaps
Capability flags that describe the vertex buffer and identify if the vertex buffer can contain optimized vertex data. This parameter can be any combination of the following flags:

(none)
The vertex buffer should be created in whatever memory the driver chooses to allow efficient read operations.

D3DVBCAPS_OPTIMIZED
The vertex buffer contains optimized vertex data. (This flag is not used when creating a new vertex buffer.)

D3DVBCAPS_SYSTEMMEMORY
The vertex buffer should be created in system memory. Use this capability for vertex buffers that will be rendered by using software devices (MMX and RGB devices).

D3DVBCAPS_WRITEONLY
Hints to the system that the application will only write to the vertex buffer. Using this flag enables the driver to choose the best memory location for efficient write operations and rendering. Attempts to read from a vertex buffer that is created with this capability can result in degraded performance.

dwFVF
A combination of flexible vertex format flags that describes the vertex format of the vertices in this buffer.

dwNumVertices
The maximum number of vertices that this vertex buffer can contain.

Remarks

Software devices—MMX and RGB devices—cannot render from a video memory (local or non-local) vertex buffer. To render a vertex buffer using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.
Header: Declared in d3dtypes.h.

See Also

Vertex Buffer Descriptions, Vertex Buffers

D3DVIEWPORT

[This is preliminary documentation and subject to change.]

The **D3DVIEWPORT** structure defines the visible 3-D volume and the 2-D screen area that a 3-D volume projects onto for the **IDirect3DViewport3::GetViewport** and **IDirect3DViewport3::SetViewport** methods.

For the **IDirect3D2** and **IDirect3DDevice2** interfaces, this structure has been superseded by the **D3DVIEWPORT2** structure.

```
typedef struct _D3DVIEWPORT {
    DWORD    dwSize;
    DWORD    dwX;
    DWORD    dwY;
    DWORD    dwWidth;
    DWORD    dwHeight;
    D3DVALUE dvScaleX;
    D3DVALUE dvScaleY;
    D3DVALUE dvMaxX;
    D3DVALUE dvMaxY;
    D3DVALUE dvMinZ;
    D3DVALUE dvMaxZ;
} D3DVIEWPORT, *LPD3DVIEWPORT;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

dwX and **dwY**
Coordinates of the top-left corner of the viewport.

dwWidth and **dwHeight**
Dimensions of the viewport.

dvScaleX and **dvScaleY**
Values of the **D3DVALUE** type describing how coordinates are scaled. The relevant coordinates here are the nonhomogeneous coordinates that result from the perspective division that projects the vertices onto the w=1 plane.

dvMaxX, **dvMaxY**, **dvMinZ**, and **dvMaxZ**

Values of the **D3DVALUE** type describing the maximum and minimum nonhomogeneous coordinates of x, y, and z. Again, the relevant coordinates are the nonhomogeneous coordinates that result from the perspective division.

Remarks

When the viewport is changed, the driver builds a new transformation matrix.

The coordinates and dimensions of the viewport are given relative to the top left of the device.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DVALUE, **IDirect3DViewport3::GetViewport**, **IDirect3DViewport3::SetViewport**

D3DVIEWPORT2

[This is preliminary documentation and subject to change.]

The **D3DVIEWPORT2** structure defines the visible 3-D volume and the window dimensions that a 3-D volume projects onto. This structure is used by the methods of the **IDirect3D2** and **IDirect3DDevice2** interfaces, and in particular by the **IDirect3DViewport3::GetViewport2** and **IDirect3DViewport3::SetViewport2** methods. This structure was introduced in DirectX 5.0.

```
typedef struct _D3DVIEWPORT2 {
    DWORD      dwSize;
    DWORD      dwX;
    DWORD      dwY;
    DWORD      dwWidth;
    DWORD      dwHeight;
    D3DVALUE   dvClipX;
    D3DVALUE   dvClipY;
    D3DVALUE   dvClipWidth;
    D3DVALUE   dvClipHeight;
    D3DVALUE   dvMinZ;
    D3DVALUE   dvMaxZ;
} D3DVIEWPORT2, *LPD3DVIEWPORT2;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

dwX and **dwY**
Pixel coordinates of the top-left corner of the viewport on the render target surface. Unless you want to render to a subset of the surface, these members can be set to zero.

dwWidth and **dwHeight**
Dimensions of the viewport on the render target surface, in pixels. Unless you are rendering only to a subset of the surface, these members should be set to the dimensions of the render target surface.

dvClipX and **dvClipY**
Coordinates of the top-left corner of the clipping volume.

The relevant coordinates here are the nonhomogeneous coordinates that result from the perspective division that projects the vertices onto the w=1 plane.

dvClipWidth and **dvClipHeight**
Dimensions of the clipping volume projected onto the w=1 plane. Unless you want to render to a subset of the surface, these members can be set to the width and height of the destination surface.

dvMinZ and **dvMaxZ**
Values of the **D3DVALUE** type describing the maximum and minimum nonhomogeneous z-coordinates resulting from the perspective divide and projected onto the w=1 plane. The values in these members must not be identical.

Remarks

The **dwX**, **dwY**, **dwWidth** and **dwHeight** members describe the position and dimensions of the viewport on the render target surface. Usually, applications render to the entire target surface; when rendering on a 640x480 surface, these members should be 0, 0, 640, and 480, respectively.

The **dvClipX**, **dvClipY**, **dvClipWidth**, **dvClipHeight**, **dvMinZ**, and **dvMaxZ** members define the non-normalized post-perspective 3-D view volume which is visible to the viewer. In most cases, **dvClipX** is set to -1.0 and **dvClipY** is set to the inverse of the viewport's aspect ratio on the target surface, which can be calculated by dividing the **dwHeight** member by **dwWidth**. Similarly, the **dvClipWidth** member is typically 2.0 and **dvClipHeight** is set to twice the aspect ratio set in **dwClipY**. The **dvMinZ** and **dvMaxZ** are usually set to 0.0 and 1.0.

Unlike the **D3DVIEWPORT** structure, **D3DVIEWPORT2** specifies the relationship between the size of the viewport and the window. The coordinates and dimensions of the viewport are given relative to the top left of the device; values increase in the y-direction as you descend the screen.

When the viewport is changed, the driver builds a new transformation matrix.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DVALUE, **IDirect3DViewport3::GetViewport2**, **IDirect3DViewport3::SetViewport2**, Clipping Volumes, Viewports and Clipping

Enumerated Types

[This is preliminary documentation and subject to change.]

This section contains information about the following enumerated types used with Direct3D Immediate Mode.

- **D3DANTIALIASMODE**
- **D3DBLEND**
- **D3DCMPFUNC**
- **D3DCULL**
- **D3DFILLMODE**
- **D3DFOGMODE**
- **D3DLIGHTSTATETYPE**
- **D3DLIGHTTYPE**
- **D3DOPCODE**
- **D3DPRIMITIVETYPE**
- **D3DRENDERSTATETYPE**
- **D3DSHADEMODE**
- **D3DSTENCILOP**
- **D3DTEXTUREADDRESS**
- **D3DTEXTUREBLEND**
- **D3DTEXTUREFILTER**
- **D3DTEXTUREMAGFILTER**
- **D3DTEXTUREMINFILTER**
- **D3DTEXTUREMIPFILTER**
- **D3DTEXTUREOP**
- **D3DTEXTURESTAGESTATETYPE**
- **D3DTRANSFORMSTATETYPE**
- **D3DVERTEXTYPE**

- **D3DZBUFFERTYPE**

D3DANTIALIASMODE

[This is preliminary documentation and subject to change.]

The **D3DANTIALIASMODE** enumerated type defines the supported antialiasing mode for the D3DRENDERSTATE_ANTIALIAS value in the **D3DRENDERSTATETYPE** enumerated type. These values define the settings only for full-scene antialiasing (for more information, see Antialiasing).

```
typedef enum _D3DANTIALIASMODE {
    D3DANTIALIAS_NONE         = 0,
    D3DANTIALIAS_SORTDEPENDENT = 1,
    D3DANTIALIAS_SORTINDEPENDENT = 2
    D3DANTIALIAS_FORCE_DWORD     = 0x7fffffff,
} D3DANTIALIASMODE;
```

Members

D3DANTIALIAS_NONE
: No antialiasing is performed. This is the default setting.

D3DANTIALIAS_SORTDEPENDENT
: Antialiasing is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur.

D3DANTIALIAS_SORTINDEPENDENT
: Antialiasing is not dependent on the sort order of the polygons.

D3DANTIALIAS_FORCE_DWORD
: Forces this enumerated type to be 32 bits in size.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DBLEND

[This is preliminary documentation and subject to change.]

The **D3DBLEND** enumerated type defines the supported blend mode for the D3DRENDERSTATE_DESTBLEND values in the **D3DRENDERSTATETYPE** enumerated type. In the member descriptions that follow, the RGBA values of the source and destination are indicated with the subscripts *s* and *d*.

```
typedef enum _D3DBLEND {
    D3DBLEND_ZERO           = 1,
    D3DBLEND_ONE            = 2,
    D3DBLEND_SRCCOLOR       = 3,
    D3DBLEND_INVSRCCOLOR    = 4,
    D3DBLEND_SRCALPHA       = 5,
    D3DBLEND_INVSRCALPHA    = 6,
    D3DBLEND_DESTALPHA      = 7,
    D3DBLEND_INVDESTALPHA   = 8,
    D3DBLEND_DESTCOLOR      = 9,
    D3DBLEND_INVDESTCOLOR   = 10,
    D3DBLEND_SRCALPHASAT    = 11,
    D3DBLEND_BOTHSRCALPHA   = 12,
    D3DBLEND_BOTHINVSRCALPHA = 13,
    D3DBLEND_FORCE_DWORD    = 0x7fffffff,
} D3DBLEND;
```

Members

D3DBLEND_ZERO
Blend factor is (0, 0, 0, 0).

D3DBLEND_ONE
Blend factor is (1, 1, 1, 1).

D3DBLEND_SRCCOLOR
Blend factor is (R_s, G_s, B_s, A_s).

D3DBLEND_INVSRCCOLOR
Blend factor is ($1-R_s$, $1-G_s$, $1-B_s$, $1-A_s$).

D3DBLEND_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DBLEND_INVSRCALPHA
Blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$).

D3DBLEND_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DBLEND_INVDESTALPHA
Blend factor is ($1-A_d$, $1-A_d$, $1-A_d$, $1-A_d$).

D3DBLEND_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DBLEND_INVDESTCOLOR
Blend factor is ($1-R_d$, $1-G_d$, $1-B_d$, $1-A_d$).

D3DBLEND_SRCALPHASAT
Blend factor is (f, f, f, 1); f = $\min(A_s, 1-A_d)$.

D3DBLEND_BOTHSRCALPHA

Obsolete. For DirectX 6.0 and later, you can achieve the same affect by setting the source and destination blend factors to D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA in separate calls.

D3DBLEND_BOTHINVSRCALPHA
Source blend factor is (1-A_s, 1-A_s, 1-A_s, 1-A_s), and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden. This blend mode is supported only for the D3DRENDERSTATE_SRCBLEND render state.

D3DBLEND_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DCMPFUNC

[This is preliminary documentation and subject to change.]

The **D3DCMPFUNC** enumerated type defines the supported compare functions for the D3DRENDERSTATE_ZFUNC, D3DRENDERSTATE_ALPHAFUNC, and D3DRENDERSTATE_STENCILFUNC render states.

```
typedef enum _D3DCMPFUNC {
    D3DCMP_NEVER       = 1,
    D3DCMP_LESS        = 2,
    D3DCMP_EQUAL       = 3,
    D3DCMP_LESSEQUAL   = 4,
    D3DCMP_GREATER     = 5,
    D3DCMP_NOTEQUAL    = 6,
    D3DCMP_GREATEREQUAL = 7,
    D3DCMP_ALWAYS      = 8,
    D3DCMP_FORCE_DWORD   = 0x7fffffff,
} D3DCMPFUNC;
```

Members

D3DCMP_NEVER
Always fail the test.

D3DCMP_LESS
Accept the new pixel if its value is less than the value of the current pixel.

D3DCMP_EQUAL
Accept the new pixel if its value equals the value of the current pixel.

D3DCMP_LESSEQUAL
Accept the new pixel if its value is less than or equal to the value of the current pixel.

D3DCMP_GREATER
Accept the new pixel if its value is greater than the value of the current pixel.

D3DCMP_NOTEQUAL
Accept the new pixel if its value does not equal the value of the current pixel.

D3DCMP_GREATEREQUAL
Accept the new pixel if its value is greater than or equal to the value of the current pixel.

D3DCMP_ALWAYS
Always pass the test.

D3DCMP_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DCULL

[This is preliminary documentation and subject to change.]

The **D3DCULL** enumerated type defines the supported cull modes. These define how back faces are culled when rendering a geometry.

```
typedef enum _D3DCULL {
    D3DCULL_NONE = 1,
    D3DCULL_CW   = 2,
    D3DCULL_CCW  = 3,
    D3DCULL_FORCE_DWORD   = 0x7fffffff,
} D3DCULL;
```

Members

D3DCULL_NONE
Do not cull back faces.

D3DCULL_CW
Cull back faces with clockwise vertices.

D3DCULL_CCW
Cull back faces with counterclockwise vertices.

D3DCULL_FORCE_DWORD

Forces this enumerated type to be 32 bits in size.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DPRIMCAPS, **D3DRENDERSTATETYPE**

D3DFILLMODE

[This is preliminary documentation and subject to change.]

The **D3DFILLMODE** enumerated type contains constants describing the fill mode. These values are used by the D3DRENDERSTATE_FILLMODE render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DFILLMODE {
    D3DFILL_POINT     = 1,
    D3DFILL_WIREFRAME = 2,
    D3DFILL_SOLID     = 3
    D3DFILL_FORCE_DWORD   = 0x7fffffff,
} D3DFILLMODE;
```

Members

D3DFILL_POINT
Fill points.

D3DFILL_WIREFRAME
Fill wireframes. This fill mode currently does not work for clipped primitives when you are using the DrawPrimitive methods.

D3DFILL_SOLID
Fill solids.

D3DFILL_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DFOGMODE

[This is preliminary documentation and subject to change.]

The **D3DFOGMODE** enumerated type contains constants describing the fog mode. These values are used by the D3DRENDERSTATE_FOGTABLEMODE render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DFOGMODE {
    D3DFOG_NONE   = 0,
    D3DFOG_EXP    = 1,
    D3DFOG_EXP2   = 2,
    D3DFOG_LINEAR = 3
    D3DFOG_FORCE_DWORD   = 0x7fffffff,
} D3DFOGMODE;
```

Members

D3DFOG_NONE
No fog effect.

D3DFOG_EXP
The fog effect intensifies exponentially, according to the following formula:

$$f = 1 / e^{d \times density}$$

D3DFOG_EXP2
The fog effect intensifies exponentially with the square of the distance, according to the following formula:

$$f = 1 / e^{(d \times density)^2}$$

D3DFOG_LINEAR
The fog effect intensifies linearly between the start and end points, according to the following formula:

$$f = \frac{end - d}{end - start}$$

This is the only fog mode currently supported.

D3DFOG_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

In monochromatic (ramp) lighting mode, fog works properly only when the fog color is black. (If there is no lighting, any fog color will work, since in this case any fog color is effectively black.)

For more information about fog, see Fog.

Note

Fog can be considered a measure of visibility—the lower the fog value produced by one of the fog equations, the less visible an object is.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DLIGHTSTATETYPE

[This is preliminary documentation and subject to change.]

The **D3DLIGHTSTATETYPE** enumerated type defines the light state for the **IDirect3DDevice3::SetLightState** method. This enumerated type is part of the **D3DSTATE** structure.

```
typedef enum _D3DLIGHTSTATETYPE {
    D3DLIGHTSTATE_MATERIAL   = 1,
    D3DLIGHTSTATE_AMBIENT    = 2,
    D3DLIGHTSTATE_COLORMODEL = 3,
    D3DLIGHTSTATE_FOGMODE    = 4,
    D3DLIGHTSTATE_FOGSTART   = 5,
    D3DLIGHTSTATE_FOGEND     = 6,
    D3DLIGHTSTATE_FOGDENSITY = 7,
    D3DLIGHTSTATE_COLORVERTEX      = 8,
    D3DLIGHTSTATE_FORCE_DWORD     = 0x7fffffff,
} D3DLIGHTSTATETYPE;
```

Members

D3DLIGHTSTATE_MATERIAL

Defines the material that is lit and used to compute the final color and intensity values during rasterization. The default value is NULL. This value must be set when you use textures in ramp mode. When no material is selected (NULL), the Direct3D lighting engine is disabled.

D3DLIGHTSTATE_AMBIENT

Sets the color and intensity of the current ambient light. If an application specifies this value, it should not specify a light as a parameter. The default value is 0.

D3DLIGHTSTATE_COLORMODEL
One of the values for the **D3DCOLORMODEL** data type. The default value is D3DCOLOR_RGB.

D3DLIGHTSTATE_FOGMODE
One of the members of the **D3DFOGMODE** enumerated type. The default value is D3DFOG_NONE.

D3DLIGHTSTATE_FOGSTART
Defines the starting value for fog. The default value is 1.0.

D3DLIGHTSTATE_FOGEND
Defines the ending value for fog. The default value is 100.0.

D3DLIGHTSTATE_FOGDENSITY
Defines the density setting for fog. The default value is 1.0.

D3DLIGHTSTATE_COLORVERTEX
Enables or disables the use of the vertex color in lighting calculations for vertices whose vertex format (specified as a flexible vertex format) includes color information. The default value, TRUE, enables the use of the vertex color in lighting. Set this to FALSE to cause the system to ignore the vertex color. Per-vertex color is supported only by lights for which properties are defined by a **D3DLIGHT2** structure.

D3DLIGHTSTATE_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

When programming for execute buffers, this enumerated type is used with the D3DOP_STATELIGHT opcode.

Setting D3DLIGHTSTATE_COLORVERTEX to FALSE instructs the geometry pipeline to ignore part of each vertex (the vertex color). The only reason to use this light state is to change the appearance of the geometry without respecifying it in a different vertex format.

If D3DLIGHTSTATE_COLORVERTEX is set to TRUE and a diffuse vertex color is present, the output alpha is equal to the diffuse alpha for the vertex. Otherwise, output alpha is equal to the alpha component of diffuse material, clamped to the range [0, 255].

You can disable or enable lighting by including or omitting the D3DDP_DONOTLIGHT flag when calling a standard **IDirect3DDevice3** rendering method, such as **IDirect3DDevice3::DrawPrimitive**. If you are using vertex buffers, disable or enable lighting by ommiting or including the D3DVOP_LIGHT flag when you call the **IDirect3DVertexBuffer::ProcessVertices** method.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DOPCODE and **D3DSTATE**, Light Properties

D3DLIGHTTYPE

[This is preliminary documentation and subject to change.]

The **D3DLIGHTTYPE** enumerated type defines the light type. This enumerated type is part of the **D3DLIGHT2** structure.

```
typedef enum _D3DLIGHTTYPE {
    D3DLIGHT_POINT        = 1,
    D3DLIGHT_SPOT         = 2,
    D3DLIGHT_DIRECTIONAL  = 3,
    D3DLIGHT_PARALLELPOINT = 4,
    D3DLIGHT_FORCE_DWORD  = 0x7fffffff,
} D3DLIGHTTYPE;
```

Members

D3DLIGHT_POINT
: Light is a point source. The light has a position in space and radiates light in all directions.

D3DLIGHT_SPOT
: Light is a spotlight source. This light is something like a point light except that the illumination is limited to a cone. This light type has a direction and several other parameters which determine the shape of the cone it produces. For information about these parameters, see the **D3DLIGHT2** structure.

D3DLIGHT_DIRECTIONAL
: Light is a directional source. This is equivalent to using a point light source at an infinite distance.

D3DLIGHT_PARALLELPOINT
: Light is a parallel point source. This light type acts like a directional light except its direction is the vector going from the light position to the origin of the geometry it is illuminating.

D3DLIGHT_FORCE_DWORD
: Forces this enumerated type to be 32 bits in size.

Remarks

Directional and parallel-point lights are slightly faster than point light sources, but point lights look a little better. Spotlights offer interesting visual effects but are computationally expensive.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DOPCODE

[This is preliminary documentation and subject to change.]

The **D3DOPCODE** enumerated type contains the opcodes for execute buffer.

```
typedef enum _D3DOPCODE {
    D3DOP_POINT          = 1,
    D3DOP_LINE           = 2,
    D3DOP_TRIANGLE       = 3,
    D3DOP_MATRIXLOAD     = 4,
    D3DOP_MATRIXMULTIPLY  = 5,
    D3DOP_STATETRANSFORM  = 6,
    D3DOP_STATELIGHT     = 7,
    D3DOP_STATERENDER    = 8,
    D3DOP_PROCESSVERTICES = 9,
    D3DOP_TEXTURELOAD    = 10,
    D3DOP_EXIT           = 11,
    D3DOP_BRANCHFORWARD   = 12,
    D3DOP_SPAN           = 13,
    D3DOP_SETSTATUS      = 14,
    D3DOP_FORCE_DWORD    = 0x7fffffff,
} D3DOPCODE;
```

Members

D3DOP_POINT
Sends a point to the renderer. Operand data is described by the **D3DPOINT** structure.

D3DOP_LINE
Sends a line to the renderer. Operand data is described by the **D3DLINE** structure.

D3DOP_TRIANGLE

Sends a triangle to the renderer. Operand data is described by the **D3DTRIANGLE** structure.

D3DOP_MATRIXLOAD
Triggers a data transfer in the rendering engine. Operand data is described by the **D3DMATRIXLOAD** structure.

D3DOP_MATRIXMULTIPLY
Triggers a data transfer in the rendering engine. Operand data is described by the **D3DMATRIXMULTIPLY** structure.

D3DOP_STATETRANSFORM
Sets the value of internal state variables in the rendering engine for the transformation module. Operand data is a variable token and the new value. The token identifies the internal state variable, and the new value is the value to which that variable should be set. For more information about these variables, see the **D3DSTATE** structure and the **D3DTRANSFORMSTATETYPE** enumerated type.

D3DOP_STATELIGHT
Sets the value of internal state variables in the rendering engine for the lighting module. Operand data is a variable token and the new value. The token identifies the internal state variable, and the new value is the value to which that variable should be set. For more information about these variables, see the **D3DSTATE** structure and the **D3DLIGHTSTATETYPE** enumerated type.

D3DOP_STATERENDER
Sets the value of internal state variables in the rendering engine for the rendering module. Operand data is a variable token and the new value. The token identifies the internal state variable, and the new value is the value to which that variable should be set. For more information about these variables, see the **D3DSTATE** structure and the **D3DRENDERSTATETYPE** enumerated type.

D3DOP_PROCESSVERTICES
Sets both lighting and transformations for vertices. Operand data is described by the **D3DPROCESSVERTICES** structure.

D3DOP_TEXTURELOAD
Triggers a data transfer in the rendering engine. Operand data is described by the **D3DTEXTURELOAD** structure.

D3DOP_EXIT
Signals that the end of the list has been reached.

D3DOP_BRANCHFORWARD
Enables a branching mechanism within the execute buffer. For more information, see the **D3DBRANCH** structure.

D3DOP_SPAN
This opcode is obsolete and is silently ignored. Spans a list of points with the same y value. For more information, see the **D3DSPAN** structure.

D3DOP_SETSTATUS
Resets the status of the execute buffer. For more information, see the **D3DSTATUS** structure.

D3DOP_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

An execute buffer has two parts: an array of vertices (each typically with position, normal vector, and texture coordinates) and an array of opcode/operand groups. One opcode can have several operands following it; the system simply performs the relevant operation on each operand.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DINSTRUCTION

D3DPRIMITIVETYPE

[This is preliminary documentation and subject to change.]

The **D3DPRIMITIVETYPE** enumerated type lists the primitives supported by DrawPrimitive methods. This type was introduced in DirectX 5.0.

```
typedef enum _D3DPRIMITIVETYPE {
    D3DPT_POINTLIST     = 1,
    D3DPT_LINELIST      = 2,
    D3DPT_LINESTRIP     = 3,
    D3DPT_TRIANGLELIST  = 4,
    D3DPT_TRIANGLESTRIP = 5,
    D3DPT_TRIANGLEFAN   = 6
    D3DPT_FORCE_DWORD   = 0x7fffffff,
} D3DPRIMITIVETYPE;
```

Members

D3DPT_POINTLIST
Renders the vertices as a collection of isolated points.

D3DPT_LINELIST
Renders the vertices as a list of isolated straight line segments. Calls using this primitive type will fail if the count is less than 2, or is odd.

D3DPT_LINESTRIP

Renders the vertices as a single polyline. Calls using this primitive type will fail if the count is less than 2.

D3DPT_TRIANGLELIST
Renders the specified vertices as a sequence of isolated triangles. Each group of 3 vertices defines a separate triangle. Calls using this primitive type will fail if the count is less than 3, or if not evenly divisible by 3.

Backface culling is affected by the current winding order render state.

D3DPT_TRIANGLESTRIP
Renders the vertices as a triangle strip. Calls using this primitive type will fail if the count is less than 3. The backface removal flag is automatically flipped on even numbered triangles.

D3DPT_TRIANGLEFAN
Renders the vertices as a triangle fan. Calls using this primitive type will fail if the count is less than 3.

D3DPT_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

Using triangle strips or fans is often more effecient than using triangle lists, as fewer vertices are duplicated. For a conceptual overview and information about defining triangle strips and fans, see Triangle Strips and Triangle Fans.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice3::Begin, **IDirect3DDevice3::BeginIndexed**, **IDirect3DDevice3::DrawIndexedPrimitive**, **IDirect3DDevice3::DrawPrimitive**, Primitive Types

D3DRENDERSTATETYPE

[This is preliminary documentation and subject to change.]

The **D3DRENDERSTATETYPE** enumerated type describes the render state for the D3DOP_STATERENDER opcode. This enumerated type is part of the **D3DSTATE** structure. The values mentioned in the following descriptions are set in the second member of this structure.

Values 40 through 49 were introduced with DirectX 5.0.

```
typedef enum _D3DRENDERSTATETYPE {
    D3DRENDERSTATE_TEXTUREHANDLE      = 1,    // texture handle
    D3DRENDERSTATE_ANTIALIAS          = 2,    // antialiasing mode
    D3DRENDERSTATE_TEXTUREADDRESS     = 3,    // texture address
    D3DRENDERSTATE_TEXTUREPERSPECTIVE = 4,    // perspective correction
    D3DRENDERSTATE_WRAPU              = 5,    // wrap in u direction
    D3DRENDERSTATE_WRAPV              = 6,    // wrap in v direction
    D3DRENDERSTATE_ZENABLE            = 7,    // enable z test
    D3DRENDERSTATE_FILLMODE           = 8,    // fill mode
    D3DRENDERSTATE_SHADEMODE          = 9,    // shade mode
    D3DRENDERSTATE_LINEPATTERN        = 10,   // line pattern
    D3DRENDERSTATE_MONOENABLE         = 11,   // enable mono rendering
    D3DRENDERSTATE_ROP2               = 12,   // raster operation
    D3DRENDERSTATE_PLANEMASK          = 13,   // physical plane mask
    D3DRENDERSTATE_ZWRITEENABLE       = 14,   // enable z writes
    D3DRENDERSTATE_ALPHATESTENABLE    = 15,   // enable alpha tests
    D3DRENDERSTATE_LASTPIXEL          = 16,   // draw last pixel in a line
    D3DRENDERSTATE_TEXTUREMAG         = 17,   // how textures are magnified
    D3DRENDERSTATE_TEXTUREMIN         = 18,   // how textures are reduced
    D3DRENDERSTATE_SRCBLEND           = 19,   // blend factor for source
    D3DRENDERSTATE_DESTBLEND          = 20,   // blend factor for destination
    D3DRENDERSTATE_TEXTUREMAPBLEND    = 21,   // blend mode for map
    D3DRENDERSTATE_CULLMODE           = 22,   // back-face culling mode
    D3DRENDERSTATE_ZFUNC              = 23,   // z-comparison function
    D3DRENDERSTATE_ALPHAREF           = 24,   // reference alpha value
    D3DRENDERSTATE_ALPHAFUNC          = 25,   // alpha-comparison function
    D3DRENDERSTATE_DITHERENABLE       = 26,   // enable dithering
    D3DRENDERSTATE_ALPHABLENDENABLE   = 27,   // enable alpha blending
    D3DRENDERSTATE_FOGENABLE          = 28,   // enable fog
    D3DRENDERSTATE_SPECULARENABLE     = 29,   // enable specular highlights
    D3DRENDERSTATE_ZVISIBLE           = 30,   // enable z-checking
    D3DRENDERSTATE_SUBPIXEL           = 31,   // enable subpixel correction
    D3DRENDERSTATE_SUBPIXELX          = 32,   // enable x subpixel correction
    D3DRENDERSTATE_STIPPLEDALPHA      = 33,   // enable stippled alpha
    D3DRENDERSTATE_FOGCOLOR           = 34,   // fog color
    D3DRENDERSTATE_FOGTABLEMODE       = 35,   // fog mode
    D3DRENDERSTATE_FOGTABLESTART      = 36,   // fog table start
    D3DRENDERSTATE_FOGTABLEEND        = 37,   // fog table end
    D3DRENDERSTATE_FOGTABLEDENSITY    = 38,   // fog density
    D3DRENDERSTATE_STIPPLEENABLE      = 39,   // enables stippling
    D3DRENDERSTATE_EDGEANTIALIAS      = 40,   // antialias edges
    D3DRENDERSTATE_COLORKEYENABLE     = 41,   // enable color-key transparency
    D3DRENDERSTATE_BORDERCOLOR        = 43,   // border color
    D3DRENDERSTATE_TEXTUREADDRESSU    = 44,   // u texture address mode
    D3DRENDERSTATE_TEXTUREADDRESSV    = 45,   // v texture address mode
    D3DRENDERSTATE_MIPMAPLODBIAS      = 46,   // mipmap LOD bias
```

```
    D3DRENDERSTATE_ZBIAS            = 47,   // z-bias
    D3DRENDERSTATE_RANGEFOGENABLE   = 48,   // enables range-based fog
    D3DRENDERSTATE_ANISOTROPY       = 49,   // max. anisotropy
    D3DRENDERSTATE_FLUSHBATCH       = 50,   // explicit flush for DP batching (DX5 Only)
    D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT=51, // enable sort-independent
transparency
    D3DRENDERSTATE_STENCILENABLE    = 52,   // enable or disable stenciling
    D3DRENDERSTATE_STENCILFAIL      = 53,   // stencil operation
    D3DRENDERSTATE_STENCILZFAIL     = 54,   // stencil operation
    D3DRENDERSTATE_STENCILPASS      = 55,   // stencil operation
    D3DRENDERSTATE_STENCILFUNC      = 56,   // stencil comparison function
    D3DRENDERSTATE_STENCILREF       = 57,   // reference value for stencil test
    D3DRENDERSTATE_STENCILMASK      = 58,   // mask value used in stencil test
    D3DRENDERSTATE_STENCILWRITEMASK = 59,   // stencil buffer write mask
    D3DRENDERSTATE_TEXTUREFACTOR    = 60,   // texture factor
    D3DRENDERSTATE_STIPPLEPATTERN00 = 64,   // first line of stipple pattern
      // Stipple patterns 01 through 30 omitted here.
    D3DRENDERSTATE_STIPPLEPATTERN31 = 95,   // last line of stipple pattern
    D3DRENDERSTATE_WRAP0            = 128,  // wrapping flags for first texture
      // Wrap renderstates 1 through 6 omitted here.
    D3DRENDERSTATE_WRAP7            = 135,  // wrapping flags for last texture
    D3DRENDERSTATE_FORCE_DWORD      = 0x7fffffff,
} D3DRENDERSTATETYPE;
```

Members

D3DRENDERSTATE_TEXTUREHANDLE

Texture handle for use when rendering with the **IDirect3DDevice2** or earlier interfaces. The default value is NULL, which disables texture mapping and reverts to flat or Gouraud shading.

If the specified texture is in a system memory surface and the driver can only support texturing from display memory surfaces, the call will fail.

In retail builds the texture handle is not validated.

D3DRENDERSTATE_ANTIALIAS

One of the members of the **D3DANTIALIASMODE** enumerated type specifying the desired type of full-scene antialiasing. The default value is D3DANTIALIAS_NONE. For more information, see Full-scene Antialiasing and Antialiasing States.

You can only enable full-scene antialiasing on devices that expose the D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT or D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT capabilities.

D3DRENDERSTATE_TEXTUREADDRESS

This render state is superseded by the D3DTSS_ADDRESS texture stage state value set through the **IDirect3DDevice3::SetTextureStageState** method, but can still be used to set the addressing mode of the first texture stage. Valid values are

members of the **D3DTEXTUREADDRESS** enumerated type. The default value is D3DTADDRESS_WRAP. For more information, see Texture Addressing Modes.

Applications that need to specify separate texture-addressing modes for the u and v coordinates of a texture can use the D3DRENDERSTATE_TEXTUREADDRESSU and D3DRENDERSTATE_TEXTUREADDRESSV render states.

D3DRENDERSTATE_TEXTUREPERSPECTIVE
TRUE to enable for perspective correct texture mapping. (See perspective correction.) For the IDirect3DDevice3 interface, the default value is TRUE. For earlier interfaces, the default is FALSE. For more information, see Texture Perspective State.

D3DRENDERSTATE_WRAPU and
D3DRENDERSTATE_WRAPV
These render states are superseded by the D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7 render states, but can be used to set wrapping for the first texture stage. Set to TRUE for wrapping in u direction. The default value is FALSE. For more information, see Texture Wrapping.

D3DRENDERSTATE_ZENABLE
The depth buffering state, as one of the members of the **D3DZBUFFERTYPE** enumerated type. Set this state to D3DZB_TRUE to enable z-buffering, D3DZB_USEW to enable w-buffering, or D3DZB_FALSE to disable depth buffering.

The default value for this render state is D3DZB_TRUE if a depth buffer is attached to the render-target surface, and D3DZB_FALSE otherwise.

D3DRENDERSTATE_FILLMODE
One or more members of the **D3DFILLMODE** enumerated type. The default value is D3DFILL_SOLID.

D3DRENDERSTATE_SHADEMODE
One or more members of the **D3DSHADEMODE** enumerated type. The default value is D3DSHADE_GOURAUD.

D3DRENDERSTATE_LINEPATTERN
The **D3DLINEPATTERN** structure. The default values are 0 for **wRepeatPattern** and 0 for **wLinePattern**.

D3DRENDERSTATE_MONOENABLE
TRUE to enable monochromatic rendering, using a gray scale based on the blue channel of the color rather than full RGB. The default value is FALSE. If the device does not support RGB rendering, the value will be TRUE. Applications can check whether the device supports RGB rendering by using the **dcmColorModel** member of the **D3DDEVICEDESC** structure.

In monochromatic rendering, only the intensity (gray scale) component of the color and specular components are interpolated across the triangle. This means that only one channel (gray) is interpolated across the triangle instead of 3 channels (R,G,B), which is a performance gain for some hardware. This gray-

scale component is derived from the blue channel of the color and specular components of the triangle.

D3DRENDERSTATE_ROP2
One of the 16 standard Windows ROP2 binary raster operations specifying how the supplied pixels are combined with the pixels of the display surface. The default value is R2_COPYPEN. Applications can use the D3DPRASTERCAPS_ROP2 flag in the **dwRasterCaps** member of the **D3DPRIMCAPS** structure to determine whether additional raster operations are supported.

D3DRENDERSTATE_PLANEMASK
Physical plane mask whose type is **ULONG**. The default value is the bitwise negation of zero (~0). This physical plane mask can be used to turn off the red bit, the blue bit, and so on. This render state is not supported by the software rasterizers, and is often ignored by hardware drivers. To disable writes to the color buffer by using alpha blending, you can set D3DRENDERSTATE_SRCBLEND to D3DBLEND_ZERO and D3DRENDERSTATE_DESTBLEND to D3DBLEND_ONE.

D3DRENDERSTATE_ZWRITEENABLE
TRUE to enable writes to the depth buffer. The default value is TRUE. This member enables an application to prevent the system from updating the depth buffer with new depth values. If this state is FALSE, depth comparisons are still made according to the render state D3DRENDERSTATE_ZFUNC (assuming depth buffering is taking place), but depth values are not written to the buffer.

D3DRENDERSTATE_ALPHATESTENABLE
TRUE to enable alpha tests. The default value is FALSE. This member enables applications to turn off the tests that otherwise would accept or reject a pixel based on its alpha value.

The incoming alpha value is compared with the reference alpha value using the comparison function provided by the D3DRENDERSTATE_ALPHAFUNC render state. When this mode is enabled, alpha blending occurs only if the test succeeds.

D3DRENDERSTATE_LASTPIXEL
FALSE to enable drawing the last pixel in a line or triangle. The default value is TRUE.

D3DRENDERSTATE_TEXTUREMAG
This render state is superseded by the D3DTSS_MAGFILTER texture stage stage, set through the **IDirect3DDevice3::SetTextureStageState** method, but can still be used to set the magnification filter for the first texture stage. This render state can be one of the members of the **D3DTEXTUREFILTER** enumerated type, which describes how a texture should be filtered when it is being magnified (that is, when a texel must cover more than one pixel). The valid values are D3DFILTER_NEAREST (the default) and D3DFILTER_LINEAR.

D3DRENDERSTATE_TEXTUREMIN
This render state is superseded by the D3DTSS_MINFILTER texture stage stage, set through the **IDirect3DDevice3::SetTextureStageState** method, but can still

be used to set the minification filter for the first texture stage. This render state can be one of the members of the **D3DTEXTUREFILTER** enumerated type, which describes how a texture should be filtered when it is being made smaller (that is, when a pixel contains more than one texel). Any of the members of the **D3DTEXTUREFILTER** enumerated type can be specified for this render state. The default value is D3DFILTER_NEAREST.

D3DRENDERSTATE_SRCBLEND
One of the members of the **D3DBLEND** enumerated type. The default value is D3DBLEND_ONE.

D3DRENDERSTATE_DESTBLEND
One of the members of the **D3DBLEND** enumerated type. The default value is D3DBLEND_ZERO.

D3DRENDERSTATE_TEXTUREMAPBLEND
This render state is used when rendering with the **IDirect3DDevice2** interface. When rendering multiple textures with the **IDirect3DDevice3** interface, you can set blending operations and arguments through the **IDirect3DDevice3::SetTextureStageState** method. For more information, see Multiple Texture Blending.

One of the members of the **D3DTEXTUREBLEND** enumerated type. The default value is D3DTBLEND_MODULATE.

D3DRENDERSTATE_CULLMODE
Specifies how back-facing triangles are to be culled, if at all. This can be set to one of the members of the **D3DCULL** enumerated type. The default value is D3DCULL_CCW.

D3DRENDERSTATE_ZFUNC
One of the members of the **D3DCMPFUNC** enumerated type. The default value is D3DCMP_LESSEQUAL. This member enables an application to accept or reject a pixel based on its distance from the camera.

The depth value of the pixel is compared with the depth buffer value. If the depth value of the pixel passes the comparison function, the pixel is written.

The depth value is written to the depth buffer only if the render state is TRUE.

Software rasterizers and many hardware accelerators work faster if the depth test fails, since there is no need to filter and modulate the texture if the pixel is not going to be rendered.

D3DRENDERSTATE_ALPHAREF
Value specifying a reference alpha value against which pixels are tested when alpha-testing is enabled. This is an 8 bit value placed in the low 8 bits of the DWORD render state value. Values can range from 0x00000000 to 0x000000FF.

D3DRENDERSTATE_ALPHAFUNC
One of the members of the **D3DCMPFUNC** enumerated type. The default value is D3DCMP_ALWAYS. This member enables an application to accept or reject a pixel based on its alpha value.

D3DRENDERSTATE_DITHERENABLE
TRUE to enable dithering. The default value is FALSE.

D3DRENDERSTATE_ALPHABLENDENABLE

TRUE to enable alpha-blended transparency. The default value is FALSE. This member supersedes the legacy D3DRENDERSTATE_BLENDENABLE render state; see remarks for more information.

Prior to DirectX 5.0, the software rasterizers used this render state to toggle both color keying and alpha blending. Currently, you can use the D3DRENDERSTATE_COLORKEYENABLE render state to toggle color keying. (Hardware rasterizers have always used the D3DRENDERSTATE_BLENDENABLE render state only for toggling alpha blending.)

The type of alpha blending is determined by the D3DRENDERSTATE_SRCBLEND and D3DRENDERSTATE_DESTBLEND render states. D3DRENDERSTATE_ALPHABLENDENABLE, with D3DRENDERSTATE_COLORKEYENABLE, allows fine blending control.

D3DRENDERSTATE_ALPHABLENDENABLE does not affect the texture-blending modes specified by the **D3DTEXTUREBLEND** enumerated type. Texture blending is logically well before the D3DRENDERSTATE_ALPHABLENDENABLE part of the pixel pipeline. The only interaction between the two is that the alpha portions remaining in the polygon after the **D3DTEXTUREBLEND** phase may be used in the D3DRENDERSTATE_ALPHABLENDENABLE phase to govern interaction with the content in the frame buffer.

Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **D3DDEVICEDESC** structure to find out whether this render state is supported.

D3DRENDERSTATE_FOGENABLE

TRUE to enable fog blending. The default value is FALSE. For more information, see Fog Blending and Fog.

D3DRENDERSTATE_SPECULARENABLE

TRUE to enable specular highlights. For the **IDirect3DDevice3** interface, the default value is FALSE. For earlier interfaces, the default is TRUE.

Specular highlights are calculated as though every vertex in the object being lit were at the object's origin. This gives the expected results as long as the object is modeled around the origin and the distance from the light to the object is relatively large.

D3DRENDERSTATE_ZVISIBLE

This render state is not supported.

D3DRENDERSTATE_SUBPIXEL

TRUE to enable subpixel correction. The default value is FALSE.

Subpixel correction is the ability to draw pixels in precisely their correct locations. In a system that implemented subpixel correction, if a pixel were at position 0.1356, its position would be interpolated from the actual coordinate rather than simply drawn at 0 (using the integer values). Hardware can be non-subpixel correct or subpixel correct in x or in both x and y. When interpolating across the x-direction the actual coordinate is used. All hardware should be

subpixel correct. Some software rasterizers are not subpixel correct because of the performance loss.

Subpixel correction means that the hardware always pre-steps the interpolant values in the x-direction to the nearest pixel centers and then steps one pixel at a time in the y-direction. For each x span it also pre-steps in the x-direction to the nearest pixel center and then steps in the x-direction one pixel each time. This results in very accurate rendering and eliminates almost all jittering of pixels on triangle edges. Most hardware either doesn't support it (always off) or always supports it (always on).

D3DRENDERSTATE_SUBPIXELX
TRUE to enable subpixel correction in the x-direction only. The default value is FALSE.

D3DRENDERSTATE_STIPPLEDALPHA
TRUE to enable stippled alpha. The default value is FALSE.

Current software rasterizers ignore this render state. You can use the D3DPSHADECAPS_ALPHAFLATSTIPPLED flag in the **D3DPRIMCAPS** structure to discover whether the current hardware supports this render state.

D3DRENDERSTATE_FOGCOLOR
Value whose type is **D3DCOLOR**. The default value is 0. For more information, see Fog Color.

D3DRENDERSTATE_FOGTABLEMODE
The fog formula to be used for pixel fog. Set to one of the members of the **D3DFOGMODE** enumerated type. The default value is D3DFOG_NONE. For more information, see Pixel Fog.

D3DRENDERSTATE_FOGTABLESTART
D3DRENDERSTATE_FOGTABLEEND
Depth at which pixel fog effects begin and end for linear fog mode. Depth is specified in world-space for hardware devices (which use eye-relative fog) or in device-space for software devices. For more information, see Pixel Fog Parameters and Eye-Relative vs. Z-Based Depth.

These render states enable you to exclude fog effects for positions close to the camera. For example, you could set the starting depth to 0.3 to prevent fog effects for depths between 0.0 and 0.299, and the ending depth to 0.7 to prevent additional fog effects for depths between 0.701 and 1.0.

D3DRENDERSTATE_FOGTABLEDENSITY
Fog density for pixel fog to be used in the exponential fog modes (D3DFOG_EXP and D3DFOG_EXP2). Valid density values range from 0.0 to 1.0, inclusive. The default value is 1.0. For more information, see Pixel Fog Parameters.

D3DRENDERSTATE_STIPPLEENABLE
Enables stippling in the device driver. When stippled alpha is enabled, it overrides the current stipple pattern, as specified by the D3DRENDERSTATE_STIPPLEPATTERN00 through D3DRENDERSTATE_STIPPLEPATTERN31 render states. When stippled alpha is disabled, the stipple pattern must be returned.

D3DRENDERSTATE_EDGEANTIALIAS

TRUE to antialias lines forming the convex outline of objects. The default value is FALSE. For more information, see Edge Antialiasing and Antialiasing States. When set to TRUE, applications should only render lines, and only to the exterior edges of polygons in a scene. The behavior is undefined if triangles or points are drawn when this render state is set. Antialiasing is performed simply by averaging the values of neighboring pixels. Although this is not the best way to perform antialiasing, it can be very efficient; hardware that supports this kind of operation is becoming more common.

You can only enable edge antialiasing on devices that expose the D3DPRASTERCAPS_ANTIALIASEDGES capability.

D3DRENDERSTATE_COLORKEYENABLE

TRUE to enable color-keyed transparency. The default value is FALSE. You can use this render state with D3DRENDERSTATE_ALPHABLENDENABLE to implement fine blending control.

Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **D3DDEVICEDESC** structure to find out whether this render state is supported.

When color-keyed transparency is enabled, only texture surfaces that were created with the DDSD_CKSRCBLT flag will be affected. Surfaces that were created without the DDSD_CKSRCBLT flag will exhibit color-keyed transparency effects.

D3DRENDERSTATE_BORDERCOLOR

A **DWORD** value specifying a border color. If the texture addressing mode is specified as D3DTADDRESS_BORDER (as set in the **D3DTEXTUREADDRESS** enumerated type), this render state specifies the border color the system uses when it encounters texture coordinates outside the range [0.0, 1.0].

The format of the physical-color information specified by the **DWORD** value depends on the format of the DirectDraw surface.

D3DRENDERSTATE_TEXTUREADDRESSU

This render state is superseded by the D3DTSS_ADDRESSU texture stage state value set through the **IDirect3DDevice3::SetTextureStageState** method, but can still be used to set the addressing mode of the first texture stage. Valid values are members of the **D3DTEXTUREADDRESS** enumerated type. The default value is D3DTADDRESS_WRAP. For more information, see Texture Addressing Modes.

This render state applies only to the u texture coordinate. This render state, along with D3DRENDERSTATE_TEXTUREADDRESSV, allows you to specify separate texture-addressing modes for the u and v coordinates of a texture. Because the D3DRENDERSTATE_TEXTUREADDRESS render state applies to both the u and v texture coordinates, it overrides any values set for the D3DRENDERSTATE_TEXTUREADDRESSU render state.

D3DRENDERSTATE_TEXTUREADDRESSV

This render state is superseded by the D3DTSS_ADDRESSV texture stage state value set through the **IDirect3DDevice3::SetTextureStageState** method, but can still be used to set the addressing mode of the first texture stage. Valid values are members of the **D3DTEXTUREADDRESS** enumerated type. The default value is D3DTADDRESS_WRAP. For more information, see Texture Addressing Modes.

This render state applies only to the v texture coordinate. This render state, along with D3DRENDERSTATE_TEXTUREADDRESSU, allows you to specify separate texture-addressing modes for the u and v coordinates of a texture. Because the D3DRENDERSTATE_TEXTUREADDRESS render state applies to both the u and v texture coordinates, it overrides any values set for the D3DRENDERSTATE_TEXTUREADDRESSV render state.

D3DRENDERSTATE_MIPMAPLODBIAS

Floating-point **D3DVALUE** value used to change the level of detail (LOD) bias. This value offsets the value of the mipmap level that is computed by trilinear texturing. It is usually in the range –1.0 to 1.0; the default value is 0.0.

Each unit bias (+/-1.0) biases the selection by exactly one mipmap level. A positive bias will cause the use of larger mipmap levels, resulting in a sharper but more aliased image. A negative bias will cause the use of smaller mipmap levels, resulting in a blurrier image. Applying a negative bias also results in the referencing of a smaller amount of texture data, which can boost performance on some systems.

D3DRENDERSTATE_ZBIAS

An integer value in the range 0 to 16 that causes polygons that are physically coplanar to appear separate. Polygons with a high z-bias value will appear in front of polygons with a low value, without requiring sorting for drawing order. Polygons with a value of 1 appear in front of polygons with a value of 0, and so on. The default value is zero. For more information, see Using Depth Buffers.

D3DRENDERSTATE_RANGEFOGENABLE

TRUE to enable range-based vertex fog. (The default value is FALSE, in which case the system uses depth-based fog.) In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene. In range-based fog, all fog methods work as usual, except that they use range instead of depth in the computations.

Range is the correct factor to use for fog computations, but depth is commonly used instead because range is expensive to compute and depth is generally already available. Using depth to calculate fog has the undesirable effect of having the 'fogginess' of peripheral objects change as the eye is rotated — in this case, the depth changes while the range remains constant.

This render state works only with **D3DVERTEX** vertices. When you specify **D3DLVERTEX** or **D3DTLVERTEX** vertices, the F (fog) component of the RGBF fog value should already be corrected for range.

Since no hardware currently supports per-pixel range-based fog, range correction offered only for vertex fog. For more information, see Range-based Fog and Vertex Fog.

D3DRENDERSTATE_ANISOTROPY

This render state is superseded by the D3DTSS_MAXANISOTROPY texture stage state, set through the **IDirect3DDevice3::SetTextureStageState** method, but can still be used to set the degree of anisotropic filtering for the first texture stage.

This render state can be an integer value that enables a degree of anisotropic filtering, used for bilinear or trilinear filtering. The value determines the maximum aspect ratio of the sampling filter kernel. To determine the range of appropriate values, use the D3DPRASTERCAPS_ANISOTROPY flag in the **D3DPRIMCAPS** structure.

Anisotropy is the distortion visible in the texels of a 3-D object whose surface is oriented at an angle with respect to the plane of the screen. The anisotropy is measured as the elongation (length divided by width) of a screen pixel that is inverse-mapped into texture space.

D3DRENDERSTATE_FLUSHBATCH

Flush any pending DrawPrimitive batches. When rendering with texture handles (using the **IDirect3DDevice2** interface) you must flush batched primitives after modifying the current texture surface. Batched primitives are implicitly flushed when rendering with the **IDirect3DDevice3** interface, as well as when rendering with execute buffers.

D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT

TRUE to enable sort-independent transparency, or FALSE to disable.

D3DRENDERSTATE_STENCILENABLE

TRUE to enable stenciling, or FALSE to disable stenciling. The default value is FALSE. For more information, see Stencil Buffers.

D3DRENDERSTATE_STENCILFAIL

Stencil operation to perform if the stencil test fails. This can be one of the members of the **D3DSTENCILOP** enumerated type. The default value is D3DSTENCILOP_KEEP. For more information, see Stencil Buffers.

D3DRENDERSTATE_STENCILZFAIL

Stencil operation to perform if the stencil test passes and depth test (z-test) fails. This can be one of the members of the **D3DSTENCILOP** enumerated type. The default value is D3DSTENCILOP_KEEP. For more information, see Stencil Buffers.

D3DRENDERSTATE_STENCILPASS

Stencil operation to perform if both the stencil and depth (z) tests pass. This can be one of the members of the **D3DSTENCILOP** enumerated type. The default value is D3DSTENCILOP_KEEP. For more information, see Stencil Buffers.

D3DRENDERSTATE_STENCILFUNC

Comparison function for the stencil test. This can be one of the members of the **D3DCMPFUNC** enumerated type. The default value is D3DCMP_ALWAYS.

The comparison function is used to compare the reference value to a stencil buffer entry. This comparison only applies to the bits in the reference value and stencil buffer entry that are set in the stencil mask (set by the

D3DRENDERSTATE_STENCILMASK render state). If the comparison is true, the stencil test passes.

D3DRENDERSTATE_STENCILREF
Integer reference value for the stencil test. The default value is 0.

D3DRENDERSTATE_STENCILMASK
Mask applied to the reference value and each stencil buffer entry to determine the significant bits for the stencil test. The default mask is 0xFFFFFFFF.

D3DRENDERSTATE_STENCILWRITEMASK
Write mask applied to values written into the stencil buffer. The default mask is 0xFFFFFFFF.

D3DRENDERSTATE_TEXTUREFACTOR
Color used for multiple texture blending with the D3DTA_TFACTOR texture-blending argument or **D3DTOP_BLENDFACTORALPHA** texture-blending operation. The associated value is a **D3DCOLOR** variable.

D3DRENDERSTATE_STIPPLEPATTERN00 through
D3DRENDERSTATE_STIPPLEPATTERN31
Stipple pattern. Each render state applies to a separate line of the stipple pattern. Together, these render states specify a 32x32 stipple pattern.

D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7
Texture wrapping behavior for multiple textures. Valid values for these render states are a combination of one or both of the D3DWRAP_U and D3DWRAP_V flags, which cause the system to wrap in the u and v directions for a given texture coordinate set. The default value for these render states is 0 (wrapping disabled in both directions). For more information, see Texture Wrapping.

D3DRENDERSTATE_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

The D3DRENDERSTATE_BLENDENABLE member was superseded by the D3DRENDERSTATE_ALPHABLENDENABLE member. Its name was changed to make its meaning more explicit. To maintain compatibility with legacy applications, the D3DRENDERSTATE_BLENDENABLE constant is declared as equivalent to D3DRENDERSTATE_ALPHABLENDENABLE:

```
#define D3DRENDERSTATE_BLENDENABLE  D3DRENDERSTATE_ALPHABLENDENABLE
```

Direct3D defines the D3DRENDERSTATE_WRAPBIAS constant as a convenience for applications to enable or disable texture wrapping based on the zero-based integer of a texture coordinate set (rather than explicitly using one of the D3DRENDERSTATE_WRAP*n* state values). Add the D3DRENDERSTATE_WRAPBIAS value to the zero-based index of a texture coordinate set to calculate the D3DRENDERSTATE_WRAP*n* value that corresponds to that index, as shown in the following example:

```
// Enable U/V wrapping for textures that use the texture
```

```
// coordinate set at the index within the dwIndex variable.
HRESULT hr = lpD3DDevice->SetRenderState(
    dwIndex + D3DRENDERSTATE_WRAPBIAS,
    D3DWRAP_U | D3DWRAPV);

// If dwIndex is 3, the value that results from
// the addition equates to D3DRENDERSTATE_WRAP3 (131).
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DOPCODE, **D3DSTATE**

D3DSHADEMODE

[This is preliminary documentation and subject to change.]

The **D3DSHADEMODE** enumerated type describes the supported shade mode for the D3DRENDERSTATE_SHADEMODE render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DSHADEMODE {
    D3DSHADE_FLAT         = 1,
    D3DSHADE_GOURAUD      = 2,
    D3DSHADE_PHONG        = 3,
    D3DSHADE_FORCE_DWORD  = 0x7fffffff,
} D3DSHADEMODE;
```

Members

D3DSHADE_FLAT
: Flat shade mode. The color and specular component of the first vertex in the triangle are used to determine the color and specular component of the face. These colors remain constant across the triangle; that is, they aren't interpolated.

D3DSHADE_GOURAUD
: Gouraud shade mode. The color and specular components of the face are determined by a linear interpolation between all three of the triangle's vertices.

D3DSHADE_PHONG
: Phong shade mode is not currently supported.

D3DSHADE_FORCE_DWORD

Forces this enumerated type to be 32 bits in size.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DRENDERSTATETYPE

D3DSTENCILOP

[This is preliminary documentation and subject to change.]

The **D3DSTENCILOP** enumerated type describes the stencil operations for the D3DRENDERSTATE_STENCILFAIL, D3DRENDERSTATE_STENCILZFAIL, D3DRENDERSTATE_STENCILPASS render states.

```
typedef enum _D3DSTENCILOP {
    D3DSTENCILOP_KEEP         = 1,
    D3DSTENCILOP_ZERO         = 2,
    D3DSTENCILOP_REPLACE      = 3,
    D3DSTENCILOP_INCRSAT      = 4,
    D3DSTENCILOP_DECRSAT      = 5,
    D3DSTENCILOP_INVERT       = 6,
    D3DSTENCILOP_INCR         = 7,
    D3DSTENCILOP_DECR         = 8,
    D3DSTENCILOP_FORCE_DWORD  = 0x7fffffff
} D3DSTENCILOP;
```

Members

D3DSTENCILOP_KEEP
Do not update the entry in the stencil buffer. This is the default value.

D3DSTENCILOP_ZERO
Set the stencil-buffer entry to zero.

D3DSTENCILOP_REPLACE
Replace the stencil-buffer entry with reference value.

D3DSTENCILOP_INCRSAT
Increment the stencil-buffer entry, clamping to the maximum value. See remarks for infomation on the maximum stencil-buffer values.

D3DSTENCILOP_DECRSAT
Decrement the stencil-buffer entry, clamping to zero.

D3DSTENCILOP_INVERT
Invert the bits in the stencil-buffer entry.

D3DSTENCILOP_INCR
Increment the stencil-buffer entry, wrapping to zero if the new value exceeds the maximum value. See remarks for infomation on the maximum stencil-buffer values.

D3DSTENCILOP_DECR
Decrement the stencil-buffer entry, wrapping to the maximum value if the new value is less than zero.

D3DSTENCILOP_FORCE_DWORD
Forces this enumeration to be compiled to 32 bits in size. This value is not used.

Remarks

Stencil-buffer entries are integer values ranging inclusively from 0 to 2^n -1, where *n* is the bit depth of the stencil buffer.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DRENDERSTATETYPE, Stencil Buffers

D3DTEXTUREADDRESS

[This is preliminary documentation and subject to change.]

The **D3DTEXTUREADDRESS** enumerated type describes the supported texture addressing modes when setting them with **IDirect3DDevice3::SetTextureStageState** or with the D3DRENDERSTATE_TEXTUREADDRESS render state.

```
typedef enum _D3DTEXTUREADDRESS {
    D3DTADDRESS_WRAP         = 1,
    D3DTADDRESS_MIRROR       = 2,
    D3DTADDRESS_CLAMP        = 3,
    D3DTADDRESS_BORDER       = 4,
    D3DTADDRESS_FORCE_DWORD  = 0x7fffffff,
} D3DTEXTUREADDRESS;
```

Members

D3DTADDRESS_WRAP
: Tile the texture at every integer junction. For example, for u values between 0 and 3, the texture will be repeated three times; no mirroring is performed.

D3DTADDRESS_MIRROR
: Similar to D3DTADDRESS_WRAP, except that the texture is flipped at every integer junction. For u values between 0 and 1, for example, the texture is addressed normally, between 1 and 2 the texture is flipped (mirrored), between 2 and 3 the texture is normal again, and so on.

D3DTADDRESS_CLAMP
: Texture coordinates outside the range [0.0, 1.0] are set to the texture color at 0.0 or 1.0, respectively.

D3DTADDRESS_BORDER
: Texture coordinates outside the range [0.0, 1.0] are set to the border color, which is a new render state corresponding to D3DRENDERSTATE_BORDERCOLOR in the **D3DRENDERSTATETYPE** enumerated type.

D3DTADDRESS_FORCE_DWORD
: Forces this enumerated type to be 32 bits in size.

Remarks

For information about using the D3DRENDERSTATE_WRAPU and D3DRENDERSTATE_WRAPV render states, see Textures.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DRENDERSTATETYPE

D3DTEXTUREBLEND

[This is preliminary documentation and subject to change.]

The **D3DTEXTUREBLEND** enumerated type defines the supported texture-blending modes. This enumerated type is used by the D3DRENDERSTATE_TEXTUREMAPBLEND render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DTEXTUREBLEND {
    D3DTBLEND_DECAL         = 1,
```

```
    D3DTBLEND_MODULATE      = 2,
    D3DTBLEND_DECALALPHA    = 3,
    D3DTBLEND_MODULATEALPHA = 4,
    D3DTBLEND_DECALMASK     = 5,
    D3DTBLEND_MODULATEMASK  = 6,
    D3DTBLEND_COPY          = 7,
    D3DTBLEND_ADD           = 8,
    D3DTBLEND_FORCE_DWORD   = 0x7fffffff,
} D3DTEXTUREBLEND;
```

Members

D3DTBLEND_DECAL

Decal texture-blending mode is supported. In this mode, the RGB and alpha values of the texture replace the colors that would have been used with no texturing.

cPix = cTex

aPix = aTex

D3DTBLEND_MODULATE

Modulate texture-blending mode is supported. In this mode, the RGB values of the texture are multiplied with the RGB values that would have been used with no texturing. Any alpha values in the texture replace the alpha values in the colors that would have been used with no texturing; if the texture does not contain an alpha component, alpha values at the vertices in the source are interpolated between vertices.

*cPix = cSrc * cTex*

if(the texture has an alpha channel*)*

aPix = aTex

else

aPix = aSrc

D3DTBLEND_DECALALPHA

Decal-alpha texture-blending mode is supported. In this mode, the RGB and alpha values of the texture are blended with the colors that would have been used with no texturing, according to the following formulas:

*cPix = (cSrc * (1.0 - aTex)) + (aTex * cTex)*

aPix = aSrc

D3DTBLEND_MODULATEALPHA

Modulate-alpha texture-blending mode is supported. In this mode, the RGB values of the texture are multiplied with the RGB values that would have been used with no texturing, and the alpha values of the texture are multiplied with the alpha values that would have been used with no texturing.

*cPix = cSrc * cTex*

*aPix = aSrc * aTex*

D3DTBLEND_DECALMASK
This blending mode is not supported.

cPix = lsb(aTex) ? cTex : cSrc
aPix = aSrc

When the least-significant bit of the texture's alpha component is zero, the effect is as if texturing were disabled.

D3DTBLEND_MODULATEMASK
This blending mode is not supported.

*cPix = lsb(aTex) ? cTex * cSrc : cSrc*
aPix = aSrc

When the least-significant bit of the texture's alpha component is zero, the effect is as if texturing were disabled.

D3DTBLEND_COPY
This blending mode is obsolete, and is treated as equivalent to the D3DTBLEND_DECAL texture-blending mode.

D3DTBLEND_ADD
Add the Gouraud interpolants to the texture lookup with saturation semantics (that is, if the color value overflows it is set to the maximum possible value). This member was introduced in DirectX 5.0.

cPix = cTex + cSrc
aPix = aSrc

D3DTBLEND_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

In the formulas given for the members of this enumerated type, the placeholders have the following meanings:

- *cTex* is the color of the source texel
- *aTex* is the alpha component of the source texel
- *cSrc* is the interpolated color of the source primitive
- *aSrc* is the alpha component of the source primitive
- *cPix* is the new blended color value
- *aPix* is the new blended alpha value

Modulation combines the effects of lighting and texturing. Because colors are specified as values between and including 0 and 1, modulating (multiplying) the texture and preexisting colors together typically produces colors that are less bright

than either source. The brightness of a color component is undiminished when one of the sources for that component is white (1). The simplest way to ensure that the colors of a texture do not change when the texture is applied to an object is to ensure that the object is white (1,1,1).

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DTEXTUREFILTER

[This is preliminary documentation and subject to change.]

The **D3DTEXTUREFILTER** enumerated type defines the supported texture filter modes used by the D3DRENDERSTATE_TEXTUREMAG render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DTEXTUREFILTER {
    D3DFILTER_NEAREST         = 1,
    D3DFILTER_LINEAR          = 2,
    D3DFILTER_MIPNEAREST      = 3,
    D3DFILTER_MIPLINEAR       = 4,
    D3DFILTER_LINEARMIPNEAREST = 5,
    D3DFILTER_LINEARMIPLINEAR  = 6,
    D3DFILTER_FORCE_DWORD    = 0x7fffffff,
} D3DTEXTUREFILTER;
```

Members

D3DFILTER_NEAREST
: The texel with coordinates nearest to the desired pixel value is used. This is a point filter with no mipmapping.

  This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DFILTER_LINEAR
: A weighted average of a 2×2 area of texels surrounding the desired pixel is used. This is a bilinear filter with no mipmapping.

  This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DFILTER_MIPNEAREST
: The closest mipmap level is chosen and a point filter is applied.

D3DFILTER_MIPLINEAR
: The closest mipmap level is chosen and a bilinear filter is applied within it.

D3DFILTER_LINEARMIPNEAREST
The two closest mipmap levels are chosen and then a linear blend is used between point filtered samples of each level.

D3DFILTER_LINEARMIPLINEAR
The two closest mipmap levels are chosen and then combined using a bilinear filter.

D3DFILTER_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

All of these filter modes are valid with the D3DRENDERSTATE_TEXTUREMIN render state, but only the first two (D3DFILTER_NEAREST and D3DFILTER_LINEAR) are valid with D3DRENDERSTATE_TEXTUREMAG.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DTEXTUREMAGFILTER

[This is preliminary documentation and subject to change.]

The **D3DTEXTUREMAGFILTER** enumerated type defines texture magnification filtering modes for a texture stage.

```
typedef enum _D3DTEXTUREMAGFILTER {
    D3DTFG_POINT        = 1,
    D3DTFG_LINEAR       = 2,
    D3DTFG_FLATCUBIC    = 3,
    D3DTFG_GAUSSIANCUBIC= 4,
    D3DTFG_ANISOTROPIC  = 5,
    D3DTFG_FORCE_DWORD  = 0x7fffffff,
} D3DTEXTUREMAGFILTER;
```

Members

D3DTFG_POINT
Point filtering. The texel with coordinates nearest to the desired pixel value is used.

D3DTFG_LINEAR
Bilinear interpolation filtering. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

D3DTFG_FLATCUBIC
Not currently supported; do not use.

D3DTFG_GAUSSIANCUBIC
Not currently supported; do not use.

D3DTFG_ANISOTROPIC
Anisotropic texture filtering. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

D3DTFG_FORCE_DWORD
Forces this enumerated type to compile to 32 bits in size.

Remarks

You set a texture stage's magnification filter by calling the **IDirect3DDevice3::SetTextureStageState** method with the D3DTSS_MAGFILTER value as the second parameter, and one of members of this enumeration as the third parameter.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DTEXTUREMINFILTER, **D3DTEXTUREMIPFILTER**, Texture Filtering

D3DTEXTUREMINFILTER

[This is preliminary documentation and subject to change.]

The **D3DTEXTUREMINFILTER** enumerated type defines texture minification filtering modes for a texture stage.

```
typedef enum _D3DTEXTUREMINFILTER {
    D3DTFN_POINT        = 1,
    D3DTFN_LINEAR       = 2,
    D3DTFN_ANISOTROPIC  = 3,
    D3DTFN_FORCE_DWORD  = 0x7fffffff,
} D3DTEXTUREMINFILTER;
```

Members

D3DTFN_POINT

Point filtering. The texel with coordinates nearest to the desired pixel value is used.

D3DTFN_LINEAR
Bilinear interpolation filtering. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

D3DTFN_ANISOTROPIC
Anisotropic texture filtering. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

D3DTFN_FORCE_DWORD
Forces this enumerated type to compile to 32 bits in size.

Remarks

You set a texture stage's magnification filter by calling the **IDirect3DDevice3::SetTextureStageState** method with the D3DTSS_MINFILTER value as the second parameter, and one of members of this enumeration as the third parameter.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DTEXTUREMAGFILTER, **D3DTEXTUREMIPFILTER**, Texture Filtering

D3DTEXTUREMIPFILTER

[This is preliminary documentation and subject to change.]

The **D3DTEXTUREMIPFILTER** enumerated type defines texture mipmap filtering modes for a texture stage.

```
typedef enum _D3DTEXTUREMIPFILTER {
    D3DTFP_NONE         = 1,
    D3DTFP_POINT        = 2,
    D3DTFP_LINEAR       = 3,
    D3DTFP_FORCE_DWORD  = 0x7fffffff,
} D3DTEXTUREMIPFILTER;
```

Members

D3DTFP_NONE

Mipmapping disabled. The rasterizer should use the magnification filter instead.

D3DTFP_POINT
Nearest point mipmap filtering. The rasterizer uses the color from the texel of the nearest mipmap texture.

D3DTFP_LINEAR
Trilinear mipmap interpolation. The rasterizer linearly interpolates pixel color using the texels of the two nearest mipmap textures.

D3DTFP_FORCE_DWORD
Forces this enumerated type to compile to 32 bits in size.

Remarks

You set a texture stage's magnification filter by calling the **IDirect3DDevice3::SetTextureStageState** method with the D3DTSS_MIPFILTER value as the second parameter, and one of members of this enumeration as the third parameter.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DTEXTUREMAGFILTER, **D3DTEXTUREMIPFILTER**, Texture Filtering

D3DTEXTUREOP

[This is preliminary documentation and subject to change.]

The **D3DTEXTUREOP** enumerated type defines per-stage texture blending operations. The members of this type are used when setting color or alpha operations by using the D3DTSS_COLOROP or D3DTSS_ALPHAOP values with the **IDirect3DDevice3::SetTextureStageState** method.

```
typedef enum _D3DTEXTUREOP {
    D3DTOP_DISABLE    = 1,
    D3DTOP_SELECTARG1 = 2,
    D3DTOP_SELECTARG2 = 3,
    D3DTOP_MODULATE   = 4,
    D3DTOP_MODULATE2X = 5,
    D3DTOP_MODULATE4X = 6,
    D3DTOP_ADD        = 7,
    D3DTOP_ADDSIGNED  = 8,
    D3DTOP_ADDSIGNED2X  =  9,
```

```
    D3DTOP_SUBTRACT     = 10,
    D3DTOP_ADDSMOOTH    = 11,
    D3DTOP_BLENDDIFFUSEALPHA    = 12,
    D3DTOP_BLENDTEXTUREALPHA    = 13,
    D3DTOP_BLENDFACTORALPHA     = 14,
    D3DTOP_BLENDTEXTUREALPHAPM  = 15,
    D3DTOP_BLENDCURRENTALPHA    = 16,
    D3DTOP_PREMODULATE          = 17,
    D3DTOP_MODULATEALPHA_ADDCOLOR = 18,
    D3DTOP_MODULATECOLOR_ADDALPHA = 19,
    D3DTOP_MODULATEINVALPHA_ADDCOLOR = 20,
    D3DTOP_MODULATEINVCOLOR_ADDALPHA = 21,
    D3DTOP_BUMPENVMAP           = 22,
    D3DTOP_BUMPENVMAPLUMINANCE  = 23,
    D3DTOP_DOTPRODUCT3          = 24,
    D3DTOP_FORCE_DWORD = 0x7fffffff,
  } D3DTEXTUREOP;
```

Members

Control members

D3DTOP_DISABLE

Disables output from this texture stage and all stages with a higher index. To disable texture mapping, set this operation for the first texture stage (stage 0).

D3DTOP_SELECTARG1

Use this texture stage's first color or alpha argument, unmodified, as the output. This operation affects the color argument when used with the D3DTSS_COLOROP texture stage state, and the alpha argument when used with D3DTSS_ALPHAOP.

$S_{RGBA} = Arg1$

D3DTOP_SELECTARG2

Use this texture stage's second color or alpha argument, unmodified, as the output. This operation affects the color argument when used with the D3DTSS_COLOROP texture stage state, and the alpha argument when used with D3DTSS_ALPHAOP.

$S_{RGBA} = Arg2$

Modulation members

D3DTOP_MODULATE

Multiply the components of the arguments together.

$S_{RGBA} = Arg1 \times Arg2$

D3DTOP_MODULATE2X

Multiply the components of the arguments and shift the products to the left one bit (effectively multiplying them by two) for brightening.

$S_{RGBA} = (Arg1 \times Arg2) << 1$

D3DTOP_MODULATE4X
Multiply the components of the arguments and shift the products to the left two bits (effectively multiplying them by four) for brightening.

$S_{RGBA} = (Arg1 \times Arg2) << 2$

Addition and Subtraction members
D3DTOP_ADD
Add the components of the arguments.

$S_{RGBA} = Arg1 + Arg2$

D3DTOP_ADDSIGNED
Add the components of the arguments with a -0.5 bias, making the effective range of values from -0.5 to 0.5.

$S_{RGBA} = Arg1 + Arg2 - 0.5$

D3DTOP_ADDSIGNED2X
Add the components of the arguments with a -0.5 bias, and shift the products to the left one bit.

$S_{RGBA} = (Arg1 + Arg2 - 0.5) << 2$

D3DTOP_SUBTRACT
Subtract the components of the second argument from those of the first argument.

$S_{RGBA} = Arg1 - Arg2$

D3DTOP_ADDSMOOTH
Add the first and second arguments, then subract their product from the sum.

$S_{RGBA} = Arg\ 1 + Arg2 - Arg1 \times Arg2$
$= Arg1 + Arg2\ (1 - Arg1)$

Linear alpha blending members
D3DTOP_BLENDDIFFUSEALPHA
D3DTOP_BLENDTEXTUREALPHA
D3DTOP_BLENDFACTORALPHA
D3DTOP_BLENDCURRENTALPHA
Linearly blend this texture stage using the interpolated alpha from each vertex (D3DTOP_BLENDDIFFUSEALPHA), alpha from this stage's texture (D3DTOP_BLENDTEXTUREALPHA), a scalar alpha (D3DTOP_BLENDFACTORALPHA) set with the

D3DRENDERSTATE_TEXTUREFACTOR render state, or the alpha taken from the previous texture stage (D3DTOP_BLENDCURRENTALPHA).

$S_{RGBA} = Arg\ 1 \times (Alpha) + Arg2 \times (1 - Alpha)$

D3DTOP_BLENDTEXTUREALPHAPM
Linearly blend a texture stage that uses premultiplied alpha.

$S_{RGBA} = Arg\ 1 + Arg2 \times (1 - Alpha)$

Specular mapping members

D3DTOP_PREMODULATE
Modulate this texture stage with the next texture stage.

D3DTOP_MODULATEALPHA_ADDCOLOR
Modulate the second argument's color using the first argument's alpha, then add the result to argument one. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = Arg\ 1_{RGB} + Arg1_{A} \times Arg2_{RGB}$

D3DTOP_MODULATECOLOR_ADDALPHA
Modulate the arguments, then add the first argument's alpha. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = Arg\ 1_{RGB} \times Arg2_{RGB} + Arg1_{A}$

D3DTOP_MODULATEINVALPHA_ADDCOLOR
Similar to D3DTOP_MODULATEALPHA_ADDCOLOR, but use the inverse of the first argument's alpha. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = (1 - Arg\ 1_{A}) \times Arg2_{RGB} + Arg1_{RGB}$

D3DTOP_MODULATEINVCOLOR_ADDALPHA
Similar to D3DTOP_MODULATECOLOR_ADDALPHA, but use the inverse of the first argument's color. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = (1 - Arg\ 1_{RGB}) \times Arg2_{RGB} + Arg1_{A}$

Bump mapping members

D3DTOP_BUMPENVMAP
Perform per-pixel bump-mapping using the environment map in the next texture stage (without luminance). This operation is supported only for color operations (D3DTSS_COLOROP).

D3DTOP_BUMPENVMAPLUMINANCE

Perform per-pixel bump-mapping using the environment map in the next texture stage (with luminance). This operation is supported only for color operations (D3DTSS_COLOROP).

D3DTOP_DOTPRODUCT3
Modulate the components of each argument (as signed components), add their products, then replicate the sum to all color channels, including alpha. This operation is supported for color and alpha operations.

$$S_{RGBA} = (Arg\ 1_R \times Arg2_R + Arg1_G \times Arg2_G + Arg1_B \times Arg2_B)$$

Miscellaneous member
D3DTOP_FORCE_DWORD
Forces this enumerated type to be compiled to 32 bits in size. This value is not used.

Remarks

In the preceding formulas, S_{RGBA} is the RGBA color produced by a texture operation, and *Arg1* and *Arg2* represent the complete RGBA color of the texture arguments. Individual components of an argument are shown with subscripts. For example, the alpha component for argument one would be shown as $Arg1_A$.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice3::GetTextureStageState, **IDirect3DDevice3::SetTextureStageState**, **D3DTEXTURESTAGESTATETYPE**

D3DTEXTURESTAGESTATETYPE

[This is preliminary documentation and subject to change.]

The **D3DTEXTURESTAGESTATETYPE** enumerated type defines texture stage states. Members of this enumerated type are used with the **IDirect3DDevice3::GetTextureStageState** and **IDirect3DDevice3::SetTextureStageState** methods to retrieve and set texture state values.

```
typedef enum _D3DTEXTURESTAGESTATETYPE {
    D3DTSS_COLOROP        = 1,
    D3DTSS_COLORARG1      = 2,
    D3DTSS_COLORARG2      = 3,
```

```
    D3DTSS_ALPHAOP       = 4,
    D3DTSS_ALPHAARG1     = 5,
    D3DTSS_ALPHAARG2     = 6,
    D3DTSS_BUMPENVMAT00  = 7,
    D3DTSS_BUMPENVMAT01  = 8,
    D3DTSS_BUMPENVMAT10  = 9,
    D3DTSS_BUMPENVMAT11  = 10,
    D3DTSS_TEXCOORDINDEX = 11,
    D3DTSS_ADDRESS       = 12,
    D3DTSS_ADDRESSU      = 13,
    D3DTSS_ADDRESSV      = 14,
    D3DTSS_BORDERCOLOR   = 15,
    D3DTSS_MAGFILTER     = 16,
    D3DTSS_MINFILTER     = 17,
    D3DTSS_MIPFILTER     = 18,
    D3DTSS_MIPMAPLODBIAS = 19,
    D3DTSS_MAXMIPLEVEL   = 20,
    D3DTSS_MAXANISOTROPY = 21,
    D3DTSS_BUMPENVLSCALE = 22,
    D3DTSS_BUMPENVLOFFSET = 23,
    D3DTSS_FORCE_DWORD   = 0x7fffffff,
} D3DTEXTURESTAGESTATETYPE;
```

Members

D3DTSS_COLOROP
: The texture stage state is a texture color blending operation identified by one of the members of the **D3DTEXTUREOP** enumerated type. The default value for the first texture stage (stage zero) is D3DTOP_MODULATE, and for all other stages the default is D3DTOP_DISABLE.

D3DTSS_COLORARG1
: The texture stage state is the first color argument for the stage, identified by a texture argument flag. The default argument is D3DTA_TEXTURE.

D3DTSS_COLORARG2
: The texture stage state is the second color argument for the stage, identified by a texture argument flag. The default argument is D3DTA_CURRENT.

D3DTSS_ALPHAOP
: The texture stage state is texture alpha blending operation identified by one of the members of the **D3DTEXTUREOP** enumerated type. The default value for the first texture stage (stage zero) is D3DTOP_SELECTARG1, and for all other stages the default is D3DTOP_DISABLE.

D3DTSS_ALPHAARG1
: The texture stage state is the first alpha argument for the stage, identified by a texture argument flag. The default argument is D3DTA_TEXTURE. If no texture is set for this stage, the default argument is D3DTA_DIFFUSE.

D3DTSS_ALPHAARG2
The texture stage state is the second alpha argument for the stage, identified by a texture argument flag. The default argument is D3DTA_CURRENT.

D3DTSS_BUMPENVMAT00
The texture stage state is a **D3DVALUE** for the [0][0] coefficient in a bump mapping matrix. The default value is zero.

D3DTSS_BUMPENVMAT01
The texture stage state is a **D3DVALUE** for the [0][1] coefficient in a bump mapping matrix. The default value is 0.

D3DTSS_BUMPENVMAT10
The texture stage state is a **D3DVALUE** for the [1][0] coefficient in a bump mapping matrix. The default value is 0.

D3DTSS_BUMPENVMAT11
The texture stage state is a **D3DVALUE** for the [1][1] coefficient in a bump mapping matrix. The default value is 0.

D3DTSS_TEXCOORDINDEX
Index of the texture coordinate set to use with this texture stage. The default index is 0. Set this state to the zero-based index of the coordinate set for each vertex that this texture stage will use. (You can specify up to eight sets of texture coordinates per vertex.) If a vertex does not include a set of texture coordinates at the specified index, the system defaults to using the u, v coordinates (0,0).

D3DTSS_ADDRESS
Member of the **D3DTEXTUREADDRESS** enumerated type. Selects the texture addressing method for both the u and v coordinates. The default is D3DTADDRESS_WRAP.

D3DTSS_ADDRESSU
Member of the **D3DTEXTUREADDRESS** enumerated type. Selects the texture addressing method for the u coordinate. The default is D3DTADDRESS_WRAP.

D3DTSS_ADDRESSV
Member of the **D3DTEXTUREADDRESS** enumerated type. Selects the texture addressing method for the v coordinate. The default value is D3DTADDRESS_WRAP.

D3DTSS_BORDERCOLOR
D3DCOLOR value that describes the color to be used for rasterizing texture coordinates outside the [0.0,1.0] range. The default color is 0x00000000.

D3DTSS_MAGFILTER
Member of the **D3DTEXTUREMAGFILTER** enumerated type that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFG_POINT.

D3DTSS_MINFILTER
Member of the **D3DTEXTUREMINFILTER** enumerated type that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFN_POINT.

D3DTSS_MIPFILTER

Member of the **D3DTEXTUREMIPFILTER** enumerated type that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFP_NONE.

D3DTSS_MIPMAPLODBIAS
Level of detail bias for mipmaps. Can be used to make textures appear more chunky or more blurred. The default value is 0.

D3DTSS_MAXMIPLEVEL
Maximum mipmap level-of-detail that the application will allow, expressed as an index from the top of the mipmap chain. (Lower values identify higher levels of detail within the mipmap chain). Zero, which is the default, indicates that all levels can be used. Non-zero values indicate that the application does not want to display mipmaps that have a higher level-of-detail than the mipmap at the specified index.

D3DTSS_MAXANISOTROPY
Maximum level of anisotropy. The default value is 1.

D3DTSS_BUMPENVLSCALE
D3DVALUE scale for bump map luminance. The default value is 0.

D3DTSS_BUMPENVLOFFSET
D3DVALUE offset for bump map luminance. The default value is 0.

D3DTSS_FORCE_DWORD
Forces this enumerated type to be compiled to 32 bits in size. This value is not used.

Remarks

The valid range of values for the D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT01, D3DTSS_BUMPENVMAT10, and D3DTSS_BUMPENVMAT11 bump-mapping matrix coefficients is greater than or equal to -8.0, and less than 8.0. This range, expressed in mathematical notation is [-8.0,8.0).

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DTRANSFORMSTATETYPE

[This is preliminary documentation and subject to change.]

The **D3DTRANSFORMSTATETYPE** enumerated type describes the transformation state for the D3DOP_STATETRANSFORM opcode in the **D3DOPCODE** enumerated type. This enumerated type is part of the **D3DSTATE** structure.

```
typedef enum _D3DTRANSFORMSTATETYPE {
    D3DTRANSFORMSTATE_WORLD       = 1,
    D3DTRANSFORMSTATE_VIEW        = 2,
    D3DTRANSFORMSTATE_PROJECTION  = 3,
    D3DTRANSFORMSTATE_FORCE_DWORD  = 0x7fffffff,
} D3DTRANSFORMSTATETYPE;
```

Members

D3DTRANSFORMSTATE_WORLD
: Define the matrices for the world transformation. The default value is NULL (the identity matrix).

D3DTRANSFORMSTATE_VIEW
: Define the matrices for the view transformation. The default value is NULL (the identity matrix).

D3DTRANSFORMSTATE_PROJECTION
: Define the matrices for the projection transformation. The default value is NULL (the identity matrix).

D3DTRANSFORMSTATE_FORCE_DWORD
: Forces this enumerated type to be 32 bits in size.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DOPCODE, **D3DRENDERSTATETYPE**

D3DVERTEXTYPE

[This is preliminary documentation and subject to change.]

The **D3DVERTEXTYPE** enumerated type lists the vertex types that are supported by the legacy **IDirect3DDevice2** and **IDirect3DDevice** interfaces. If your application uses **IDirect3DDevice3**, the **D3DVERTEXTYPE** enumerated type is superseded by flexible vertex format flags.

```
typedef enum _D3DVERTEXTYPE {
    D3DVT_VERTEX      = 1,
    D3DVT_LVERTEX     = 2,
    D3DVT_TLVERTEX    = 3
    D3DVT_FORCE_DWORD  = 0x7fffffff,
```

```
} D3DVERTEXTYPE;
```

Members

D3DVT_VERTEX
: All the vertices in the array are of the **D3DVERTEX** type. This setting will cause transformation, lighting and clipping to be applied to the primitive as it is rendered.

D3DVT_LVERTEX
: All the vertices in the array are of the **D3DLVERTEX** type. When used with this option, the primitive will have transformations applied during rendering.

D3DVT_TLVERTEX
: All the vertices in the array are of the **D3DTLVERTEX** type. Rasterization only will be applied to this data.

D3DVT_FORCE_DWORD
: Forces this enumerated type to be 32 bits in size.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

Vertex Formats

D3DZBUFFERTYPE

[This is preliminary documentation and subject to change.]

The **D3DZBUFFERTYPE** enumerated type describes depth-buffer formats for use with the D3DRENDERSTATE_ZENABLE render state.

```
typedef enum _D3DZBUFFERTYPE {
    D3DZB_FALSE       = 0,
    D3DZB_TRUE        = 1,
    D3DZB_USEW        = 2,
    D3DZB_FORCE_DWORD  = 0x7fffffff,
} D3DZBUFFERTYPE;
```

Members

D3DZB_FALSE
: Disable depth-buffering.

D3DZB_TRUE
Enable z-buffering.

D3DZB_USEW
Enable w-buffering. To use w-buffering, perspective-correct texture mapping must also be enabled. To enable perspective-correct texture mapping, set the D3DRENDERSTATE_TEXTUREPERSPECTIVE render state to TRUE. For DirectX 6.0, this is the default value.

D3DZB_FORCE_DWORD
Forces this enumeration to be compiled to 32-bits in size. This value is not used.

Remarks

The D3DZB_FALSE and D3DZB_TRUE values are interchangeable with the TRUE and FALSE macro values previously used with D3DRENDERSTATE_ZENABLE.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

IDirect3DDevice3::SetRenderState, Depth Buffers

Other Types

[This is preliminary documentation and subject to change.]

This section contains information about the following Direct3D Immediate Mode types that are neither structures nor enumerated types:

- **D3DCOLOR**
- **D3DCOLORMODEL**
- **D3DFIXED**
- **D3DVALUE**

D3DCOLOR

[This is preliminary documentation and subject to change.]

The **D3DCOLOR** type is the fundamental Direct3D color type.

```
typedef DWORD D3DCOLOR, D3DCOLOR, *LPD3DCOLOR;
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DRGB, **D3DRGBA**

D3DCOLORMODEL

[This is preliminary documentation and subject to change.]

The **D3DCOLORMODEL** type is used to define the color model in which the system will run. A driver can expose either or both flags in the **dcmColorModel** member of the **D3DDEVICEDESC** structure.

```
typedef DWORD D3DCOLORMODEL
```

Values

D3DCOLOR_MONO
: Use a monochromatic model (or ramp model). In this model, the blue component of a vertex color is used to define the brightness of a lit vertex.

D3DCOLOR_RGB
: Use a full RGB model.

Remarks

Prior to DirectX 5.0, these values were part of an enumerated type. The enumerated type in earlier versions of DirectX had this syntax:

```
typedef enum _D3DCOLORMODEL {
    D3DCOLOR_MONO = 1,
    D3DCOLOR_RGB  = 2,
} D3DCOLORMODEL;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

See Also

D3DDEVICEDESC, **D3DFINDDEVICESEARCH**, **D3DLIGHTSTATETYPE**

D3DFIXED

[This is preliminary documentation and subject to change.]

The **D3DFIXED** type is used to represent a 16:16 fixed point value.

```
typedef LONG D3DFIXED;
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

D3DVALUE

[This is preliminary documentation and subject to change.]

The **D3DVALUE** type is the fundamental Direct3D fractional data type.

```
typedef float D3DVALUE, *LPD3DVALUE;
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in d3dtypes.h.

Flexible Vertex Format Flags

[This is preliminary documentation and subject to change.]

Direct3D Immediate Mode uses flag values to describe vertex formats used for DrawPrimitive-based rendering. The D3dtypes.h header file defines the following flags to explicitly describe a vertex format, and provides helper macros that act as common combinations of such flags. For more information, see About Vertex Formats.

Flexible vertex format (FVF) flags
D3DFVF_DIFFUSE
Vertex format includes a diffuse color component.
D3DFVF_NORMAL

Vertex format includes a vertex normal vector. This flag cannot be used with the D3DFVF_XYZRHW flag.

D3DFVF_SPECULAR
Vertex format includes a specular color component.

D3DFVF_XYZ
Vertex format includes the position of an untransformed vertex. This flag cannot be used with the D3DFVF_XYZRHW flag. If you use this flag, you must also specify a vertex normal, a vertex color component (D3DFVF_DIFFUSE or D3DFVF_SPECULAR), or include at least one set of texture coordinates (D3DFVF_TEX1 through D3DFVF_TEX8).

D3DFVF_XYZRHW
Vertex format includes the position of a transformed vertex. This flag cannot be used with the D3DFVF_XYZ or D3DFVF_NORMAL flags. If you use this flag, you must also specify a vertex color component (D3DFVF_DIFFUSE or D3DFVF_SPECULAR) or include at least one set of texture coordinates (D3DFVF_TEX1 through D3DFVF_TEX8).

Texture-related FVF flags

D3DFVF_TEX0 through D3DFVF_TEX8
Number of texture coordinate sets for this vertex. The actual values for these flags are not sequential.

Helper macros

D3DFVF_LVERTEX
Vertex format is equivalent to the **D3DLVERTEX** vertex type.

D3DFVF_TLVERTEX
Vertex format is equivalent to the **D3DTLVERTEX** vertex type.

D3DFVF_VERTEX
Vertex format is equivalent to the **D3DVERTEX** vertex type.

Mask values

D3DFVF_POSITION_MASK
Mask for position bits.

D3DFVF_RESERVED0 and D3DFVF_RESERVED2
Mask values for reserved bits in the flexible vertex format. Do not use.

D3DFVF_RESERVED1,
This bit is reserved to indicate that the system should emulate **D3DLVERTEX** processing. If this flag is used, the D3DFVF_XYZ, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX1 flags must also be used. This equates to the effect of the **D3DFVF_LVERTEX** helper macro.

D3DFVF_TEXCOUNT_MASK
Mask value for texture flag bits.

Miscellaneous

D3DFVF_TEXCOUNT_SHIFT
The number of bits to shift an integer value that identifies the number of a texture coordinates for a vertex. This value might be used as follows:

```
DWORD dwNumTextures = 1;  // vertex has only one set of coordinates
```

```
// Shift the value for use when creating an FVF combination.
dwFVF = dwNumTextures<<D3DFVF_TEXCOUNT_SHIFT;

/*
 * Now, create an FVF combination using the shifted value.
 */
```

The following example shows some other common flag combinations:

```
// Lightweight, untransformed vertex for lit, untextured,
// Gouraud-shaded content.
dwFVF = ( D3DFVF_XYZ | D3DFVF_DIFFUSE );

// Untransformed vertex for unlit, untextured, Gouraud-shaded
// content with diffuse material color specified per vertex.
dwFVF = ( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE );

// Untransformed vertex for light-map based lighting.
dwFVF = ( D3DFVF_XYZ | D3DFVF_TEX2 );

// Transformed vertex for light-map based lighting
// with shared rhw.
dwFVF = ( D3DFVF_XYZRHW | D3DFVF_TEX2 );

// Heavyweight vertex for unlit, colored content with two
//sets of texture coordinates.
dwFVF = ( D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE |
        D3DFVF_SPECULAR | D3DFVF_TEX2 );
```

Texture Argument Flags

[This is preliminary documentation and subject to change.]

Each texture stage for a device can have two texture arguments that affect the color or alpha channel of the texture. You set and retrieve texture arguments by calling the **IDirect3DDevice3::SetTextureStageState** and **IDirect3DDevice3::GetTextureStageState**, specifying the D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAARG1 or D3DTSS_ALPHAARG2 members of the **D3DTEXTURESTAGESTATETYPE** enumerated type.

The following flags, organized as arguments and modifiers, can be used with color and alpha arguments for a texture stage. You can combine an argument flag with a modifier, but you cannot combine two argument flags.

Argument flags
D3DTA_CURRENT

The texture argument is the result of the previous blending stage. In the first texture stage (stage zero), this argument defaults to D3DTA_DIFFUSE.

D3DTA_DIFFUSE
The texture argument is the diffuse color interpolated from vertex components during Gouraud shading. If the vertex does not contain a diffuse color, the default color is 0xFFFFFFFF.

D3DTA_SELECTMASK
Mask value for all arguments; not used when setting texture arguments.

D3DTA_TEXTURE
The texture argument is the texture color for this texture stage. This is valid only for the first color and alpha arguments in a stage (the D3DTSS_COLORARG1 and D3DTSS_ALPHAARG1 members of **D3DTEXTURESTAGESTATETYPE**). If no texture is set for a stage that uses this blending argument, the system defaults to a color value of R: 1.0, G: 1.0, B: 1.0 for color, and 1.0 for alpha.

D3DTA_TFACTOR
The texture argument is the texture factor set in a previous call to the **IDirect3DDevice3::SetRenderState** with the D3DRENDERSTATE_TEXTUREFACTOR render state value.

Modifier flags

D3DTA_ALPHAREPLICATE
Replicate the alpha information to all color channels before the operation completes.

D3DTA_COMPLEMENT
Invert the argument such that, if the result of the argument were referred to by the variable *x*, the value would be 1.0 - *x*.

Return Values

[This is preliminary documentation and subject to change.]

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all Direct3D Immediate Mode methods. See the individual method descriptions for lists of the values each can return.

D3D_OK
No error occurred.

D3DERR_BADMAJORVERSION
The service you requested is unavailable in this major version of DirectX. (A "major version" denotes a primary release, such as DirectX 6.0.)

D3DERR_BADMINORVERSION
The service you requested is available in this major version of DirectX, but not in this minor version. Get the latest version of the component runtime from Microsoft. (A "minor version" denotes a secondary release, such as DirectX 6.1.)

D3DERR_COLORKEYATTACHED

The application attempted to create a texture with a surface that uses a color key for transparency.

D3DERR_CONFLICTINGTEXTUREFILTER
The current texture filters cannot be used together.

D3DERR_CONFLICTINGTEXTUREPALETTE
The current textures cannot be used simultaneously. This generally occurs when a multi-texture device requires that all palettized textures simultaneously enabled also share the same palette.

D3DERR_CONFLICTINGRENDERSTATE
The currently set render states cannot be used together.

D3DERR_DEVICEAGGREGATED
The **IDirect3DDevice3::SetRenderTarget** method was called on a device that was retrieved from the render target surface.

D3DERR_EXECUTE_CLIPPED_FAILED
The execute buffer could not be clipped during execution.

D3DERR_EXECUTE_CREATE_FAILED
The execute buffer could not be created. This typically occurs when no memory is available to allocate the execute buffer.

D3DERR_EXECUTE_DESTROY_FAILED
The memory for the execute buffer could not be deallocated.

D3DERR_EXECUTE_FAILED
The contents of the execute buffer are invalid and cannot be executed.

D3DERR_EXECUTE_LOCK_FAILED
The execute buffer could not be locked.

D3DERR_EXECUTE_LOCKED
The operation requested by the application could not be completed because the execute buffer is locked.

D3DERR_EXECUTE_NOT_LOCKED
The execute buffer could not be unlocked because it is not currently locked.

D3DERR_EXECUTE_UNLOCK_FAILED
The execute buffer could not be unlocked.

D3DERR_INITFAILED
A rendering device could not be created because the new device could not be initialized.

D3DERR_INBEGIN
The requested operation cannot be completed while scene rendering is taking place. Try again after the scene is completed and the **IDirect3DDevice::EndScene** method (or equivalent method) is called.

D3DERR_INVALID_DEVICE
The requested device type is not valid.

D3DERR_INVALIDCURRENTVIEWPORT
The currently selected viewport is not valid.

D3DERR_INVALIDMATRIX

The requested operation could not be completed because the combination of the currently set world, view, and projection matrices is invalid (the determinant of the combined matrix is zero).

D3DERR_INVALIDPALETTE
The palette associated with a surface is invalid.

D3DERR_INVALIDPRIMITIVETYPE
The primitive type specified by the application is invalid.

D3DERR_INVALIDRAMPTEXTURE
Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.

D3DERR_INVALIDVERTEXFORMAT
The combination of flexible vertex format flags specified by the application is not valid.

D3DERR_INVALIDVERTEXTYPE
The vertex type specified by the application is invalid.

D3DERR_LIGHT_SET_FAILED
The attempt to set lighting parameters for a light object failed.

D3DERR_LIGHTHASVIEWPORT
The requested operation failed because the light object is associated with another viewport.

D3DERR_LIGHTNOTINTHISVIEWPORT
The requested operation failed because the light object has not been associated with this viewport.

D3DERR_MATERIAL_CREATE_FAILED
The material could not be created. This typically occurs when no memory is available to allocate for the material.

D3DERR_MATERIAL_DESTROY_FAILED
The memory for the material could not be deallocated.

D3DERR_MATERIAL_GETDATA_FAILED
The material parameters could not be retrieved.

D3DERR_MATERIAL_SETDATA_FAILED
The material parameters could not be set.

D3DERR_MATRIX_CREATE_FAILED
The matrix could not be created. This can occur when no memory is available to allocate for the matrix.

D3DERR_MATRIX_DESTROY_FAILED
The memory for the matrix could not be deallocated.

D3DERR_MATRIX_GETDATA_FAILED
The matrix data could not be retrieved. This can occur when the matrix was not created by the current device.

D3DERR_MATRIX_SETDATA_FAILED
The matrix data could not be set. This can occur when the matrix was not created by the current device.

D3DERR_NOCURRENTVIEWPORT
The viewport parameters could not be retrieved because none have been set.

D3DERR_NOTINBEGIN
The requested rendering operation could not be completed because scene rendering has not begun. Call **IDirect3DDevice3::BeginScene** to begin rendering then try again.

D3DERR_NOVIEWPORTS
The requested operation failed because the device currently has no viewports associated with it.

D3DERR_SCENE_BEGIN_FAILED
Scene rendering could not begin.

D3DERR_SCENE_END_FAILED
Scene rendering could not be completed.

D3DERR_SCENE_IN_SCENE
Scene rendering could not begin because a previous scene was not completed by a call to the **IDirect3DDevice3::EndScene** method.

D3DERR_SCENE_NOT_IN_SCENE
Scene rendering could not be completed because a scene was not started by a previous call to the **IDirect3DDevice3::BeginScene** method.

D3DERR_SETVIEWPORTDATA_FAILED
The viewport parameters could not be set.

D3DERR_STENCILBUFFER_NOTPRESENT
The requested stencil buffer operation could not be completed because there is no stencil buffer attached to the render target surface.

D3DERR_SURFACENOTINVIDMEM
The device could not be created because the render target surface is not located in video-memory. (Hardware-accelerated devices require video-memory render target surfaces.)

D3DERR_TEXTURE_BADSIZE
The dimensions of a current texture are invalid. This can occur when an application attempts to use a texture that has non-power-of-two dimensions with a device that requires them.

D3DERR_TEXTURE_CREATE_FAILED
The texture handle for the texture could not be retrieved from the driver.

D3DERR_TEXTURE_DESTROY_FAILED
The device was unable to deallocate the texture memory.

D3DERR_TEXTURE_GETSURF_FAILED
The DirectDraw surface used to create the texture could not be retrieved.

D3DERR_TEXTURE_LOAD_FAILED
The texture could not be loaded.

D3DERR_TEXTURE_LOCK_FAILED
The texture could not be locked.

D3DERR_TEXTURE_LOCKED

The requested operation could not be completed because the texture surface is currently locked.

D3DERR_TEXTURE_NO_SUPPORT
The device does not support texture mapping.

D3DERR_TEXTURE_NOT_LOCKED
The requested operation could not be completed because the texture surface is not locked.

D3DERR_TEXTURE_SWAP_FAILED
The texture handles could not be swapped.

D3DERR_TEXTURE_UNLOCK_FAILED
The texture surface could not be unlocked.

D3DERR_TOOMANYOPERATIONS
The application is requesting more texture filtering operations than the device supports.

D3DERR_TOOMANYPRIMITIVES
The device is unable to render the provided quantity of primitives in a single pass.

D3DERR_UNSUPPORTEDALPHAARG
The device does not support one of the specified texture blending arguments for the alpha channel.

D3DERR_UNSUPPORTEDALPHAOPERATION
The device does not support one of the specified texture blending operations for the alpha channel.

D3DERR_UNSUPPORTEDCOLORARG
The device does not support the one of the specified texture blending arguments for color values.

D3DERR_UNSUPPORTEDCOLOROPERATION
The device does not support the one of the specified texture blending operations for color values.

D3DERR_UNSUPPORTEDFACTORVALUE
The specified texture factor value is not supported by the device.

D3DERR_UNSUPPORTEDTEXTUREFILTER
The specified texture filter is not supported by the device.

D3DERR_VBUF_CREATE_FAILED
The vertex buffer could not be created. This can happen when there is insufficient memory to allocate a vertex buffer.

D3DERR_VERTEXBUFFERLOCKED
The requested operation could not be completed because the vertex buffer is locked.

D3DERR_VERTEXBUFFEROPTIMIZED
The requested operation could not be completed because the vertex buffer is optimized. (The contents of optimized vertex buffers are driver specific, and considered private.)

D3DERR_VIEWPORTDATANOTSET
The requested operation could not be completed because viewport parameters have not yet been set. Set the viewport parameters by calling **IDirect3DViewport3::SetViewport** method and try again.

D3DERR_VIEWPORTHASNODEVICE
The requested operation could not be completed because the viewport has not yet been associated with a device. Associate the viewport with a rendering device by calling **IDirect3DDevice3::AddViewport** and try again.

D3DERR_WRONGTEXTUREFORMAT
The pixel format of the texture surface is not valid.

D3DERR_ZBUFF_NEEDS_SYSTEMMEMORY
The requested operation could not be completed because the specified device requires system-memory depth-buffer surfaces. (Software rendering devices require system-memory depth buffers.)

D3DERR_ZBUFF_NEEDS_VIDEOMEMORY
The requested operation could not be completed because the specified device requires video-memory depth-buffer surfaces. (Hardware-accelerated devices require video-memory depth buffers.)

D3DERR_ZBUFFER_NOTPRESENT
The requested operation could not be completed because the render target surface does not have an attached depth buffer.

Direct3D Immediate Mode Visual Basic Reference

[This is preliminary documentation and subject to change.]

This section contains reference information for the application programming interface (API) elements provided by Direct3D® Immediate Mode. Reference material is divided into the following categories:

- Classes
- Types
- Enumerations
- Flexible Vertex Format Flags
- Texture Argument Flags
- Error Codes

Classes

[This is preliminary documentation and subject to change.]

This section contains reference information for the classes provided by Direct3D Immediate Mode. The following classes are covered:

- **Direct3D3**
- **Direct3DDevice3**
- **Direct3DEnumDevices**
- **Direct3DEnumPixelFormats**
- **Direct3DLight**
- **Direct3DMaterial3**
- **Direct3DTexture2**
- **Direct3DVertexBuffer**
- **Direct3DViewport3**

Direct3D3

[#] [This is preliminary documentation and subject to change.]

Applications use the methods of the **Direct3D3** class to create Direct3D objects and set up the environment. Moreover, The **Direct3D3** class enables applications to create vertex buffers and enumerate texture map and depth-buffer formats. This section is a reference to the methods of this class. For a conceptual overview, see Direct3D Interfaces.

The **Direct3D3** class is obtained by calling the **GetDirect3D** method from a **DirectDraw4** object.

The methods of the **Direct3D3** class can be organized into the following groups:

Creation	**CreateDevice**
	CreateLight
	CreateMaterial
	CreateVertexBuffer
	CreateViewport
Enumeration	**FindDevice**
	GetDevicesEnum
	GetEnumZBufferFormats

[#] IDH__dx_Direct3D3_d3d_vb

Miscellaneous	**EvictManagedTextures**
	GetDirectDraw

See Also

Accessing Direct3D, Direct3D and DirectDraw

Direct3D3.CreateDevice

[This is preliminary documentation and subject to change.]

The **Direct3D3.CreateDevice** method creates a Direct3D device to be used with the DrawPrimitive methods.

```
object.CreateDevice( _
    guid As String, _
    surf As DirectDrawSurface4) As Direct3DDevice3
```

object
Object expression that resolves to a **Direct3D3** object.

guid
Guid for the new device. This value can be IID_Direct3DHALDevice, IID_Direct3DMMXDevice, or IID_Direct3DRGBDevice. The IID_Direct3DRampDevice, used for the ramp emulation device, is not supported by **Direct3D3**.

surf
A **DirectDrawSurface4** object for the DirectDrawSurface object that will be the device's rendering target. The surface must have been created as a 3-D device by using the DDSCAPS_3DDEVICE capability.

Return Values

If the method succeeds, the return value is a **Direct3DDevice3** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

When you call **Direct3D3.CreateDevice**, you create a device object that is separate from a DirectDraw surface object. This device uses a DirectDraw surface as a rendering target.

IDH__dx_Direct3D3.CreateDevice_d3d_vb

See Also

Direct3DDevice3

Direct3D3.CreateLight

[#] [This is preliminary documentation and subject to change.]

The **Direct3D3.CreateLight** method creates a **Direct3DLight** object. This object can then be associated with a viewport by using the **Direct3DViewport3.AddLight** method.

object.**CreateLight() As Direct3DLight**

object
Object expression that resolves to a **Direct3D3** object.

Return Values

If the method succeeds, a **Direct3DLight** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

See Also

Direct3DLight

Direct3D3.CreateMaterial

[#] [This is preliminary documentation and subject to change.]

The **Direct3D3.CreateMaterial** method allocates a **Direct3DMaterial3** object.

object.**CreateMaterial() As Direct3DMaterial3**

object
Object expression that resolves to a **Direct3D3** object.

[#] IDH__dx_Direct3D3.CreateLight_d3d_vb
[#] IDH__dx_Direct3D3.CreateMaterial_d3d_vb

Return Values

If the method succeeds, a **Direct3DMaterial3** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set. For information on trapping errors, see the Visual Basic Error Trapping topic.

Direct3D3.CreateVertexBuffer

[#] [This is preliminary documentation and subject to change.]

The **Direct3D3.CreateVertexBuffer** method creates a vertex buffer object.

object.**CreateVertexBuffer(_**
desc **As D3DVERTEXBUFFERDESC, _**
flags **As CONST_D3DDPFLAGS) As Direct3DVertexBuffer**

object
Object expression that resolves to a **Direct3D3** object.

desc
A **D3DVERTEXBUFFERDESC** type that describes the format and number of vertices that the vertex buffer will contain.

flags
One of the constants of the **CONST_D3DDPFLAGS** enumeration representing the clipping value. Set this parameter to 0 to create a vertex buffer that can contain clipping information for untransformed or transformed vertices, or use the D3DDP_DONOTCLIP flag to create a vertex buffer that will contain transformed vertices, but no clipping information.

Return Values

If the method succeeds, a **Direct3DVertexBuffer** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes:

D3DERR_INVALIDVERTEXFORMAT
D3DERR_VBUF_CREATE_FAILED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

[#] IDH__dx_Direct3D3.CreateVertexBuffer_d3d_vb

See Also

Direct3DVertexBuffer, Vertex Buffers

Direct3D3.CreateViewport

[This is preliminary documentation and subject to change.]

The **Direct3D3.CreateViewport** method creates a **Direct3DViewport2** object. The viewport is associated with a Direct3DDevice object by using the **Direct3DDevice3.AddViewport** method.

object.**CreateViewport() As Direct3DViewport2**

object
Object expression that resolves to a **Direct3D3** object.

Return Values

If the method succeeds, a **Direct3DViewport3** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Direct3D3.EvictManagedTextures

[This is preliminary documentation and subject to change.]

The **Direct3D3.EvictManagedTextures** method purges all managed textures from local or non-local video memory.

object.**EvictManagedTextures()**

object
Object expression that resolves to a **Direct3D3** object.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

IDH__dx_Direct3D3.CreateViewport_d3d_vb
IDH__dx_Direct3D3.EvictManagedTextures_d3d_vb

Remarks

This method causes Direct3D to remove any texture surfaces created with the DDSCAPS2_TEXTUREMANAGE flag from local or non-local video memory.

Direct3D3.FindDevice

[#] [This is preliminary documentation and subject to change.]

The **Direct3D3.FindDevice** method finds a device with specified characteristics and retrieves a description of it.

object.**FindDevice(_**
ds **As D3DFINDDEVICESEARCH**, **_**
findresult **As D3DFINDDEVICERESULT2)**

object
Object expression that resolves to a **Direct3D3** object.

ds
A **D3DFINDDEVICESEARCH** type describing the device to be located.

findresult
A **D3DFINDDEVICERESULT2** type describing the device if it is found.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3D3.GetDevicesEnum

[#] [This is preliminary documentation and subject to change.]

The **Direct3D3.GetDevicesEnum** method creates a **Direct3DEnumDevices** object.

object.**GetDevicesEnum() As Direct3DEnumDevices**

object
Object expression that resolves to a **Direct3D3** object.

Return Values

If the method succeeds, a **Direct3DEnumDevices** object is returned.

[#] IDH__dx_Direct3D3.FindDevice_d3d_vb
[#] IDH__dx_Direct3D3.GetDevicesEnum_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following error codes:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Direct3D3.GetDirectDraw

[This is preliminary documentation and subject to change.]

The **Direct3D3.GetDirectDraw** method creates a **DirectDraw4** object.

object.**GetDirectDraw() As DirectDraw4**

object
Object expression that resolves to a **Direct3D3** object.

Return Values

If the method succeeds, a **DirectDraw4** object is returned.

Error Codes

If the method fails, an error is raised and Err.Number may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

Direct3D3.GetEnumZBufferFormats

[This is preliminary documentation and subject to change.]

The **Direct3D3.GetEnumZBufferFormats** method creates a **Direct3DEnumPixelFormats** object.

object.**GetEnumZBufferFormats(_**
guid **As String) As Direct3DEnumPixelFormats**

object
Object expression that resolves to a **Direct3D3** object.
guid

IDH__dx_Direct3D3.GetDirectDraw_d3d_vb
IDH__dx_Direct3D3.GetEnumZBufferFormats_d3d_vb

A globally unique identifier for the device whose depth-buffer formats will be enumerated.

Return Values

If the method succeeds, a **Direct3DEnumPixelFormats** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOZBUFFERHW
DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

Direct3DDevice3

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3** class provides methods enabling applications to perform DrawPrimitive-based rendering; this is in contrast to the **Direct3DDevice** class, which applications use to work with execute buffers. You can create a Direct3DDevice object and retrieve a pointer to this object by calling the **Direct3D3.CreateDevice** method.

For a conceptual overview, see Direct3D Devices and The DrawPrimitive Methods.

The methods of the **Direct3DDevice3** class can be organized into the following groups:

Information	**GetCaps**
	GetDirect3D
	GetStats
Miscellaneous	**ComputeSphereVisibility**
	MultiplyTransform
Getting and Setting States	**GetClipStatus**
	GetCurrentViewport

[#] IDH__dx_Direct3DDevice3_d3d_vb

	GetLightState
	GetRenderState
	GetRenderTarget
	GetTransform
	SetClipStatus
	SetCurrentViewport
	SetLightState
	SetRenderState
	SetRenderTarget
	SetTransform
Rendering	**Begin**
	BeginIndexed
	DrawIndexedPrimitive
	DrawIndexedPrimitiveVB
	DrawPrimitive
	DrawPrimitiveVB
	End
	Index
	Vertex
Scenes	**BeginScene**
	EndScene
Textures	**GetTexture**
	GetTextureFormatsEnum
	GetTextureStageState
	SetTexture
	SetTextureStageState
	ValidateDevice
Viewports	**AddViewport**
	DeleteViewport
	NextViewport

This class contains methods to support more flexible vertex formats, vertex buffers, and visibility computation. This class is not intended to be used with execute buffers, and therefore does not contain any execute-buffer related methods.

See Also

Direct3D Devices, Rendering

Direct3DDevice3.AddViewport

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.AddViewport** method adds the specified viewport to the list of viewport objects associated with the device and increments the reference count of the viewport.

object.**AddViewport(***viewport* **As Direct3DViewport3)**

object
Object expression that resolves to a **Direct3DDevice3** object.

viewport
A **Direct3DViewport3** object that should be associated with this Direct3DDevice object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method will fail, returning DDERR_INVALIDPARAMS, if you attempt to add a viewport that has already been assigned to the device.

Direct3DDevice3.Begin

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.Begin** method indicates the start of a sequence of rendered primitives. This method defines the type of these primitives and the type of vertices on which they are based. The only method you can legally call between calls to **Direct3DDevice3.Begin** and **Direct3DDevice3.End** is **Direct3DDevice3.Vertex**.

object.**Begin(** _
d3dpt **As CONST_D3DPRIMITIVETYPE**, _

IDH__dx_Direct3DDevice3.AddViewport_d3d_vb
IDH__dx_Direct3DDevice3.Begin_d3d_vb

d3dvt **As CONST_D3DVERTEXTYPE**, _
flags **As Long)**

object
Object expression that resolves to a **Direct3DDevice3** object.

d3dpt
One of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration.

d3dvt
A combination of flexible vertex format flags that describe the vertex format used. Only vertices that match this description will be accepted before the corresponding **Direct3DDevice3.End**.

flags
One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP
The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT
Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS
Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice3.GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the arguments is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

This method fails if it is called after a call to the **Direct3DDevice3.Begin** or **Direct3DDevice3.BeginIndexed** method that has no bracketing call to **Direct3DDevice3.End** method. Rendering calls that specify the wrong vertex type or that perform state changes will cause rendering of this primitive to fail.

See Also

Direct3DDevice3.BeginIndexed, **Direct3DDevice3.End**, **Direct3DDevice3.Vertex**

Direct3DDevice3.BeginIndexed

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.BeginIndexed** method defines the start of a primitive based on indexing into an array of vertices. This method fails if it is called after a call to the **Direct3DDevice3.Begin** or **Direct3DDevice3.BeginIndexed** method that has no corresponding call to **Direct3DDevice3.End**. The only method you can legally call between calls to **Direct3DDevice3.BeginIndexed** and **Direct3DDevice3.End** is **Direct3DDevice3.Index**.

```
object.BeginIndexed( _
    d3Dpt As CONST_D3DPRIMITIVETYPE, _
    d3dvt As Long, _
    verts As Any, _
    vertexCount As Long, _
    flags As CONST_D3DDPFLAGS)
```

object
: **Object** expression that resolves to a **Direct3DDevice3** object.

d3Dpt
: Type of primitive to be rendered by this command. This must be one of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration. Note that the D3DPT_POINTLIST member of **CONST_D3DPRIMITIVETYPE** is not indexed.

d3dvt
: A combination of flexible vertex format flags that describe the vertex format used. Only vertices that match this description will be accepted before the corresponding **Direct3DDevice3.End**.

verts
: Pointer to the list of vertices to be used in the primitive sequence.

vertexCount
: Number of vertices in the array at *verts*.

flags
: One or more of the following constants from the **CONST_D3DDPFLAGS** enumeration defining how the primitive is drawn:

  D3DDP_DONOTCLIP
  : The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

  D3DDP_DONOTLIGHT

[#] IDH__dx_Direct3DDevice3.BeginIndexed_d3d_vb

Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS
Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice3.GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE
Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.

DDERR_INVALIDPARAMSOne of the arguments is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice3.Begin, **Direct3DDevice3.End**, **Direct3DDevice3.Index**

Direct3DDevice3.BeginScene

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.BeginScene** method begins a scene. Applications must call this method before performing any rendering, and must call **Direct3DDevice3.EndScene** when rendering is complete, and before calling **Direct3DDevice.BeginScene** again.

object.**BeginScene()**

object
Object expression that resolves to a **Direct3DDevice3** object.

[#] IDH__dx_Direct3DDevice3.BeginScene_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

See Also

Direct3DDevice3.EndScene

Direct3DDevice3.ComputeSphereVisibility

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.ComputeSphereVisibility** method calculates the visibility (complete, partial, or no visibility) of a sphere within the current viewport for this device.

```
object.ComputeSphereVisibility( _
    center As D3DVECTOR, _
    radi As Single) As Long
```

object
Object expression that resolves to a **Direct3DDevice3** object.

center
A **D3DVECTOR** type describing the center point for the sphere, in world-space coordinates.

radi
The radius for the sphere.

Return Values

If the method succeeds, the return value is a combination of the following flags from the **CONST_D3DVISFLAGS** enumeration that describe the visibility of that sphere within the current viewport for this device:

Inside flags
D3DVIS_INSIDE_BOTTOM, D3DVIS_INSIDE_FAR, D3DVIS_INSIDE_FRUSTUM, D3DVIS_INSIDE_LEFT, D3DVIS_INSIDE_NEAR, D3DVIS_INSIDE_RIGHT, D3DVIS_INSIDE_TOP
The sphere is inside the viewing frustum of the current viewport.

Intersection flags
D3DVIS_INTERSECT_BOTTOM or D3DVIS_INTERSECT_TOP
The sphere intersects the bottom or top plane of the viewing frustum for the current viewport, depending on which flag is present.

[#] IDH__dx_Direct3DDevice3.ComputeSphereVisibility_d3d_vb

D3DVIS_INTERSECT_FAR or D3DVIS_INTERSECT_NEAR
The sphere intersects the far or near plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_INTERSECT_FRUSTUM
The sphere intersects some part of the viewing frustum for the current viewport.

D3DVIS_INTERSECT_LEFT or D3DVIS_INTERSECT_RIGHT
The sphere intersects the left or right plane of the viewing frustum for the current viewport, depending on which flag is present.

Outside flags

D3DVIS_OUTSIDE_BOTTOM or D3DVIS_OUTSIDE_TOP
The sphere is outside the bottom or top plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_OUTSIDE_FAR or D3DVIS_OUTSIDE_NEAR
The sphere is outside the far or near plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_OUTSIDE_FRUSTUM
The sphere is somewhere outside the viewing frustum for the current viewport.

D3DVIS_OUTSIDE_LEFT or D3DVIS_OUTSIDE_RIGHT
The sphere is outside the left or right plane of the viewing frustum for the current viewport, depending on which flag is present.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Sphere visibility is computed by back transforming the viewing frustum to the model space, using the inverse of the combined world, view or projection matrices. If the combined matrix can not be inverted (if the determinant is zero), the method fails, returning DDERR_GENERIC.

Direct3DDevice3.DeleteViewport

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_Direct3DDevice3.DeleteViewport_d3d_vb

The **Direct3DDevice3.DeleteViewport** method removes the specified viewport from the list of viewport objects associated with the device and decrements the reference count of the viewport.

object.**DeleteViewport(***vport* **As Direct3DViewport3)**

object
Object expression that resolves to a **Direct3DDevice3** object.

vport
A **Direct3DViewport3** object of the viewport object that will be disassociated with this device.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method fails, returning DDERR_INVALIDPARAMS, if you attempt to delete a viewport from the device without previously assigning the viewport with a call to **Direct3DDevice3.AddViewport**.

See Also

Direct3DDevice3.AddViewport

Direct3DDevice3.DrawIndexedPrimitive

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.DrawIndexedPrimitive** method renders the specified geometric primitive based on indexing into an array of vertices.

object.**DrawIndexedPrimitive(** _
d3Dpt **As CONST_D3DPRIMITIVETYPE**, _
d3dvt **As CONST_D3DVERTEXTYPE**, _
vertices **As Any**, _
vertexCount **As Long**, _
indices() **As Integer**, _
indicesCount **As Long**, _

[#] IDH__dx_Direct3DDevice3.DrawIndexedPrimitive_d3d_vb

flags **As CONST_D3DDPFLAGS)**

object
Object expression that resolves to a **Direct3DDevice3** object.

d3Dpt
Type of primitive to be rendered by this command. This must be one of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration.

Note that the D3DPT_POINTLIST member of **CONST_D3DPRIMITIVETYPE** is not indexed.

d3dvt
A combination of flexible vertex format flags from the **CONST_D3DVERTEXTYPE** enumeration that describe the vertex format used.Only vertices that match this description will be accepted before the corresponding **Direct3DDevice3.End**.

vertices
Pointer to the list of vertices to be used in the primitive sequence.

vertexCount
Defines the number of vertices in the list.

Notice that this parameter is used differently from the *vertexCount* parameter in the **Direct3DDevice3.DrawPrimitive** method. In that method, the *vertexCount* parameter gives the number of vertices to draw, but here it gives the total number of vertices in the array pointed to by the *vertices* parameter. When you call **Direct3DDevice3.DrawIndexedPrimitive**, you specify the number of vertices to draw in the *indicesCount* parameter.

indices()
Pointer to an array that is to be used to index into the specified vertex list when creating the geometry to render.

IndicesCount
Specifies the number of indices provided for creating the geometry.

flags
One or more of the following constants from the **CONST_D3DDPFLAGS** defining how the primitive is drawn:

D3DDP_DONOTCLIP
The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT
Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS

Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice3.GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE
D3DERR_INVALIDPRIMITIVETYPE
D3DERR_INVALIDVERTEXTYPE
DDERR_WASSTILLDRAWING
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

In current versions of DirectX, **Direct3DDevice3.DrawIndexedPrimitive** can sometimes generate an update rectangle that is larger than it strictly needs to be. If a large number of vertices need to be processed, this can have a negative impact on the performance of your application. If you are using **D3DTLVERTEX** vertices and the system is processing more vertices than you need, you should use the D3DDP_DONOTCLIP and D3DDP_DONOTUPDATEEXTENTS flags to solve the problem.

See Also

Direct3DDevice3.DrawPrimitive, **Direct3DDevice3.DrawPrimitiveVB**, **Direct3DDevice3.DrawIndexedPrimitiveVB**

Direct3DDevice3.DrawIndexedPrimitiveVB

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.DrawIndexedPrimitiveVB** method renders a geometric primitive based on indexing into an array of vertices within a vertex buffer.

object.**DrawIndexedPrimitiveVB(_**
d3Dpt **As CONST_D3DPRIMITIVETYPE**, _
vertexBuffer **As Direct3DVertexBuffer**, _
indexArray() **As Integer**, _
indexcount **As Long**, _
flags **As CONST_D3DDPFLAGS)**

object
Object expression that resolves to a **Direct3DDevice3** object.

d3Dpt
Type of primitive to be rendered by this command. This must be one of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration.

Note that the D3DPT_POINTLIST member of **CONST_D3DPRIMITIVETYPE** is not indexed.

vertexBuffer
A **Direct3DVertexBuffer** object for the vertex buffer that contains the array of vertices. Vertices can be transformed or untransformed, optimized or unoptimized.

indexArray()
An array that will be used to index into the vertices in the vertex buffer.

indexcount
The number of indices in the array at *indexArray()*.

flags
One or more of the following constants of the **CONST_D3DDPFLAGS** enumeration defining how the primitive is drawn:

D3DDP_DONOTCLIP
The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT
Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

[#] IDH__dx_Direct3DDevice3.DrawIndexedPrimitiveVB_d3d_vb

D3DDP_DONOTUPDATEEXTENTS
Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice3.GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDPRIMITIVETYPE
D3DERR_INVALIDVERTEXTYPE
D3DERR_VERTEXBUFFERLOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_WASSTILLDRAWING

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

Software devices — MMX and RGB devices — cannot render from a video memory (local or non-local) vertex buffer. To render a vertex buffer using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

You cannot render from a locked vertex buffer; calls to the **Direct3DDevice3.DrawIndexedPrimitiveVB** or **Direct3DDevice3.DrawPrimitiveVB** method using a locked buffer will fail, returning D3DERR_VERTEXBUFFERLOCKED.

See Also

Direct3DDevice3.DrawPrimitive, **Direct3DDevice3.DrawPrimitiveVB**, **Direct3DDevice3.DrawIndexedPrimitive**

Direct3DDevice3.DrawPrimitive

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.DrawPrimitive** method renders the specified array of vertices as a sequence of geometric primitives of the specified type.

object.**DrawPrimitive(** _
d3Dpt **As CONST_D3DPRIMITIVETYPE**, _
d3dvt **As CONST_D3DVERTEXTYPE**, _
vertices **As Any**, _
vertexCount **As Long**, _
flags **As CONST_D3DDPFLAGS)**

object
Object expression that resolves to a **Direct3DDevice3** object.

d3Dpt
Type of primitive to be rendered by this command. This must be one of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration.

Note that the D3DPT_POINTLIST member of **CONST_D3DPRIMITIVETYPE** is not indexed.

d3dvt
A combination of flexible vertex format flags from the **CONST_D3DVERTEXTYPE** enumeration that describe the vertex format used. Only vertices that match this description will be accepted before the corresponding **Direct3DDevice3.End**.

vertices
The array of vertices to be used in the primitive sequence.

vertexCount
Defines the number of vertices in the array.

flags
One or more of the following constants of the **CONST_D3DDPFLAGS** enumeration defining how the primitive is drawn:

D3DDP_DONOTCLIP
The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT
Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS

[#] IDH__dx_Direct3DDevice3.DrawPrimitive_d3d_vb

Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice3.GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE
D3DERR_INVALIDPRIMITIVETYPE
D3DERR_INVALIDVERTEXTYPE
DDERR_WASSTILLDRAWING
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

See Also

Direct3DDevice3.DrawPrimitiveVB, **Direct3DDevice3.DrawIndexedPrimitive**, **Direct3DDevice3.DrawIndexedPrimitiveVB**

Direct3DDevice3.DrawPrimitiveVB

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.DrawPrimitiveVB** method renders an array of vertices in a vertex buffer as a sequence of geometric primitives.

```
object.DrawPrimitiveVB( _
    d3Dpt As CONST_D3DPRIMITIVETYPE, _
    vertexBuffer As Direct3DVertexBuffer, _
```

[#] IDH__dx_Direct3DDevice3.DrawPrimitiveVB_d3d_vb

startVertex **As Long**, _
numVertices **As Long**, _
flags **As CONST_D3DDPFLAGS)**

object
Object expression that resolves to a **Direct3DDevice3** object.

d3Dpt
Type of primitive to be rendered by this command. This must be one of the constants of the **CONST_D3DPRIMITIVETYPE** enumeration.

Note that the D3DPT_POINTLIST member of **CONST_D3DPRIMITIVETYPE** is not indexed.

vertexBuffer
A **Direct3DVertexBuffer** object for the vertex buffer that contains the array of vertices. Vertices can be transformed or untransformed, optimized or unoptimized.

startVertex
Index value of the first vertex in the primitive. The highest possible starting index is 65,535 (0xFFFF). In debug builds, specifying a starting index value that exceeds this limit will cause the method fail and return DDERR_INVALIDPARAMS.

numVertices
Number of vertices to be rendered.

flags
One or more of the following constants of the **CONST_D3DDPFLAGS** enumeration defining how the primitive is drawn:

D3DDP_DONOTCLIP
The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT
Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS
Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice3.GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card

responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE
D3DERR_INVALIDPRIMITIVETYPE
D3DERR_INVALIDVERTEXTYPE
D3DERR_VERTEXBUFFERLOCKED
DDERR_WASSTILLDRAWING
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Make sure that the vertices being rendered match the vertex format you specify. For performance reasons, Direct3D does not attempt to verify that vertex size and stride match the provided flexible vertex format. If a mismatch occurs, a memory fault is likely to result.

Software devices — MMX and RGB devices — cannot render from a video memory (local or non-local) vertex buffer. To render a vertex buffer using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

You cannot render from a locked vertex buffer; calls to the **Direct3DDevice3.DrawIndexedPrimitiveVB** or **Direct3DDevice3.DrawPrimitiveVB** method using a locked buffer will fail, returning D3DERR_VERTEXBUFFERLOCKED.

See Also

Direct3DDevice3.DrawPrimitive, **Direct3DDevice3.DrawIndexedPrimitive**, **Direct3DDevice3.DrawIndexedPrimitiveVB**

Direct3DDevice3.End

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.End** method signals the completion of a primitive sequence. This method fails if no corresponding call to the **Direct3DDevice3.Begin** method (or **Direct3DDevice3.BeginIndexed**) was made.

[#] IDH__dx_Direct3DDevice3.End_d3d_vb

object.**End()**

object
Object expression that resolves to a **Direct3DDevice3** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method fails if the vertex count is incorrect for the primitive type. It fails without drawing if it is called before a sufficient number of vertices is specified. If the number of **Direct3DDevice3.Vertex** or **Direct3DDevice3.Index** calls made is not evenly divisible by 3 (in the case of triangles), or 2 (in the case of a line list), the remainder will be ignored.

See Also

Direct3DDevice3.Begin, **Direct3DDevice3.BeginIndexed**

Direct3DDevice3.EndScene

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.EndScene** method ends a scene that was begun by calling the **Direct3DDevice3.BeginScene** method.

object.**EndScene()**

object
Object expression that resolves to a **Direct3DDevice3** object.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

[#] IDH__dx_Direct3DDevice3.EndScene_d3d_vb

Remarks

When this method succeeds, the scene will have been rendered and the device surface will hold the contents of the rendering.

You must call this method before you can call the **Direct3DDevice3.BeginScene** method to start rendering another scene, even if the previous attempt to render was unsuccessful.

See Also

Direct3DDevice3.BeginScene

Direct3DDevice3.GetCaps

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetCaps** method retrieves the capabilities of the Direct3D device.

```
object.GetCaps( _
    hwDesc As D3DDEVICEDESC, _
    helDesc As D3DDEVICEDESC)
```

object
Object expression that resolves to a **Direct3DDevice3** object.

hwDesc
A **D3DDEVICEDESC** type that will contain the hardware features of the device.

helDesc
A **D3DDEVICEDESC** type that will contain the software emulation being provided.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method does not retrieve the capabilities of the display device. To retrieve this information, use the **DirectDraw4.GetCaps** method.

[#] IDH__dx_Direct3DDevice3.GetCaps_d3d_vb

Direct3DDevice3.GetClipStatus

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetClipStatus** method gets the current clip status.

object.**GetClipStatus(***clipStatus* **As D3DCLIPSTATUS)**

object
Object expression that resolves to a **Direct3DDevice3** object.

clipStatus
A **D3DCLIPSTATUS** type that describes the current clip status.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the arguments is invalid.

See Also

Direct3DDevice3.SetClipStatus

Direct3DDevice3.GetCurrentViewport

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetCurrentViewport** method retrieves the current viewport.

object.**GetCurrentViewport() As Direct3DViewport3**

object
Object expression that resolves to a **Direct3DDevice3** object.

Return Values

If the method succeeds, a **Direct3DViewport3** object containing the current viewport will be returned. A reference is taken to the viewport object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDPARAMS	One of the arguments is invalid.
D3DERR_NOCURRENTVIEWPORT	No current viewport has been set by a call to the

IDH__dx_Direct3DDevice3.GetClipStatus_d3d_vb
IDH__dx_Direct3DDevice3.GetCurrentViewport_d3d_vb

Direct3DDevice3.SetCurrentViewport method.

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice3.SetCurrentViewport

Direct3DDevice3.GetDirect3D

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetDirect3D** method retrieves the Direct3D object for this device.

object.**GetDirect3D() As Direct3D3**

object
Object expression that resolves to a **Direct3DDevice3** object.

Return Values

If the method succeeds, a **Direct3D3** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DDevice3.GetLightState

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetLightState** method gets a single Direct3D device lighting-related state value.

object.**GetLightState(_**
state **As CONST_D3DLIGHTSTATETYPE) As Long**

object
Object expression that resolves to a **Direct3DDevice3** object.

state
Device state variable that is being queried. This parameter can be any of the constants of the **CONST_D3DLIGHTSTATETYPE** enumeration.

IDH__dx_Direct3DDevice3.GetDirect3D_d3d_vb
IDH__dx_Direct3DDevice3.GetLightState_d3d_vb

Return Values

If the method succeeds, the return value is the **Direct3DDevice** light state.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the arguments is invalid.

See Also

Direct3DDevice3.SetLightState

Direct3DDevice3.GetRenderState

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetRenderState** method gets a single Direct3DDevice rendering state parameter.

object.**GetRenderState(** _
state **As CONST_D3DRENDERSTATETYPE) As Long**

object
Object expression that resolves to a **Direct3DDevice3** object.

state
Device state variable that is being queried. This parameter can be any of the constants of the **CONST_D3DRENDERSTATETYPE** enumeration.

Return Values

If the method succeeds, the return value is the **Direct3DDevice** render state.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the arguments is invalid.

See Also

Direct3DDevice3.SetRenderState

Direct3DDevice3.GetRenderTarget

[This is preliminary documentation and subject to change.]

IDH__dx_Direct3DDevice3.GetRenderState_d3d_vb
IDH__dx_Direct3DDevice3.GetRenderTarget_d3d_vb

The **Direct3DDevice3.GetRenderTarget** method retrieves a pointer to the **DirectDrawSurface4** object that is being used as a render target.

object.**GetRenderTarget() As DirectDrawSurface4**

object
Object expression that resolves to a **Direct3DDevice3** object.

Return Values

If the method succeeds, a **DirectDrawSurface4** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set to DDERR_INVALIDPARAMS if one of the arguments is invalid.

See Also

Direct3DDevice3.SetRenderTarget

Direct3DDevice3.GetStats

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetStats** method retrieves statistics about a device.

object.**GetStats(***stat* **As D3DSTATS)**

object
Object expression that resolves to a **Direct3DDevice3** object.

stat
A **D3DSTATS** type that will be filled with the statistics.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_Direct3DDevice3.GetStats_d3d_vb

Direct3DDevice3.GetTexture

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetTexture** method retrieves a texture assigned to a given stage for a device.

object.**GetTexture(***stage* **As Long) As Direct3DTexture2**

object
Object expression that resolves to a **Direct3DDevice3** object.

stage
Stage identifier of the texture to be retrieved. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value allowed for *stage* is 7.

Return Values

If the method succeeds, the return value is the specified texture's **Direct3DTexture2** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice3.SetTexture, **Direct3DDevice3.GetTextureStageState**, **Direct3DDevice3.SetTextureStageState**

Direct3DDevice3.GetTextureFormatsEnum

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.EnumTextureFormats** method returns a **Direct3DEnumPixelFormats** object.

object.**GetTextureFormatsEnum() As Direct3DEnumPixelFormats**

IDH__dx_Direct3DDevice3.GetTexture_d3d_vb
IDH__dx_Direct3DDevice3.GetTextureFormatsEnum_d3d_vb

object
Object expression that resolves to a **Direct3DDevice3** object.

Return Values

If the method succeeds, a **Direct3DEnumPixelFormats** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Direct3DDevice3.GetTextureStageState

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetTextureStageState** method retrieves a state value for a currently assigned texture.

```
object.GetTextureStageState( _
    stage As Long, _
    state As CONST_D3DTEXTURESTAGESTATETYPE) As Long
```

object
Object expression that resolves to a **Direct3DDevice3** object.

stage
Stage identifier of the texture for which the state will be retrieved. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value allowed for *stage* is 7.

state
Texture state to be retrieved. This parameter can be any constant of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

Return Values

If the method succeeds, the return value is the state value.

[#] IDH__dx_Direct3DDevice3.GetTextureStageState_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice3.SetTextureStageState, **Direct3DDevice3.GetTexture**, **Direct3DDevice3.SetTexture**

Direct3DDevice3.GetTransform

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.GetTransform** method gets a matrix describing a transformation state.

object.**GetTransform(** _
transformType **As CONST_D3DTRANSFORMSTATETYPE**, _
matrix **As D3DMATRIX)**

object
Object expression that resolves to a **Direct3DDevice3** object.

transformType
Device state variable that is being modified. This parameter can be any of the constants of the **CONST_D3DTRANSFORMSTATETYPE** enumeration.

matrix
A **D3DMATRIX** type describing the transformation.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDPARAMS if one of the arguments is invalid.

See Also

Direct3DDevice3.SetTransform

[#] IDH__dx_Direct3DDevice3.GetTransform_d3d_vb

Direct3DDevice3.Index

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.Index** method adds a new index to the primitive sequence started with a previous call to the **Direct3DDevice3.BeginIndexed** method.

object.**Index(***vertexIndex* **As Integer)**

object
Object expression that resolves to a **Direct3DDevice3** object.

vertexIndex
Index of the next vertex to be added to the currently started primitive sequence.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice3.BeginIndexed, **Direct3DDevice3.End**

Direct3DDevice3.MultiplyTransform

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.MultiplyTransform** method multiplies a device's world, view, or projection matrices by a specified matrix. The multiplication order is *matrix* times *dstTransformStateType*.

object.**MultiplyTransform(_**
dstTransformStateType **As Long**, **_**
matrix **As D3DMATRIX)**

object
Object expression that resolves to a **Direct3DDevice3** object.

dstTransformStateType

IDH__dx_Direct3DDevice3.Index_d3d_vb
IDH__dx_Direct3DDevice3.MultiplyTransform_d3d_vb

Identifies which device matrix is to be modified. The most common setting, D3DTRANSFORMSTATE_WORLD, modifies the world matrix, but you can specify that the method modify the view or projection matrices if needed.

matrix

A **D3DMATRIX** type that modifies the current transformation.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

An application might use the **MultiplyTransform** method to work with hierarchies of transformations. For example, the geometry and transformations describing an arm might be arranged in the following hierarchy:

```
shoulder_transformation
   upper_arm geometry
   elbow transformation
      lower_arm geometry
      wrist transformation
         hand geometry
```

An application might use the following series of calls to render this hierarchy. (Not all of the parameters are shown in this pseudocode.)

```
Direct3DDevice3.SetTransform(D3DTRANSFORMSTATE_WORLD,
   shoulder_transform)
Direct3DDevice3.DrawPrimitive(upper_arm)
Direct3DDevice3.MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
   elbow_transform)
Direct3DDevice3.DrawPrimitive(lower_arm)
Direct3DDevice3.MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
   wrist_transform)
Direct3DDevice3.DrawPrimitive(hand)
```

See Also

Direct3DDevice3.DrawPrimitive, **Direct3DDevice3.SetTransform**

Direct3DDevice3.NextViewport

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.NextViewport** method enumerates the viewports associated with the device.

object.**NextViewport(_**
vp1 **As Direct3DViewport3**, **_**
flags **As CONST_D3DNEXTFLAGS) As Direct3DViewport3**

object
Object expression that resolves to a **Direct3DDevice3** object.

vp1
A **Direct3DViewport3** object of a viewport in the list of viewports associated with this Direct3D device.

flags
Flag specifying which viewport to retrieve from the list of viewports. This must be set to one of the following constants of the **CONST_D3DNEXTFLAGS** enumeration:

D3DNEXT_HEAD
Retrieve the item at the beginning of the list.

D3DNEXT_NEXT
Retrieve the next item in the list.

D3DNEXT_TAIL
Retrieve the item at the end of the list.

Return Values

If the method succeeds, a **Direct3DViewport3** object for another viewport in the device's viewport list is returned. Which viewport the method retrieves is determined by the flag in the *flags* parameter.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_Direct3DDevice3.NextViewport_d3d_vb

Remarks

If you attempt to retrieve the next viewport in the list when you are at the end of the list, this method returns Nothing.

Direct3DDevice3.SetClipStatus

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.SetClipStatus** method sets the current clip status.

object.**SetClipStatus(***clipStatus* **As D3DCLIPSTATUS)**

object
Object expression that resolves to a **Direct3DDevice3** object.

clipStatus
A **D3DCLIPSTATUS** type that describes the new settings for the clip status.

Error Codes

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice3.GetClipStatus

Direct3DDevice3.SetCurrentViewport

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.SetCurrentViewport** method sets the current viewport.

object.**SetCurrentViewport(***viewport* **As Direct3DViewport3)**

object
Object expression that resolves to a **Direct3DDevice3** object.

viewport
A **Direct3DViewport3** object for the viewport that will become the current viewport if the method is successful.

[#] IDH__dx_Direct3DDevice3.SetClipStatus_d3d_vb
[#] IDH__dx_Direct3DDevice3.SetCurrentViewport_d3d_vb

Error Codes

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Applications must call this method before calling any rendering functions. Before calling this method, applications must have already called the **Direct3DDevice3.AddViewport** method to add the viewport to the device.

Before the first call to **Direct3DDevice3.SetCurrentViewport**, the current viewport for the device is invalid, and any attempts to render using the device will fail.

See Also

Direct3DDevice3.GetCurrentViewport

Direct3DDevice3.SetLightState

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.SetLightState** method sets a single Direct3DDevice lighting-related state value.

```
object.SetLightState( _
    state As CONST_D3DLIGHTSTATETYPE, _
    lightstate As Long)
```

object
Object expression that resolves to a **Direct3DDevice3** object.

state
Device state variable that is being modified. This parameter can be any of the constants of the **CONST_D3DLIGHTSTATETYPE** enumeration.

lightstate
New value for the Direct3DDevice light state. The meaning of this parameter is dependent on the value specified for *state*. For example, if *state* were **D3DLIGHTSTATE_COLORMODEL**, the second parameter would be one of the values of the **CONST_D3DCOLORMODEL** data type.

Error Codes

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

[#] IDH__dx_Direct3DDevice3.SetLightState_d3d_vb

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Although Direct3DIM supports the D3DLIGHTSTATE_COLORVERTEX through the **Direct3DDevice3.SetLightState** method, the **Direct3DDevice3.GetLightState** method does not recognize the value, and will return DDERR_INVALIDPARAMS.

See Also

Direct3DDevice3.GetLightState, **Direct3DDevice3.SetRenderState**, **Direct3DDevice3.SetTransform**

Direct3DDevice3.SetRenderState

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.SetRenderState** method sets a single Direct3DDevice rendering state parameter.

object.**SetRenderState(** _
state **As CONST_D3DRENDERSTATETYPE**, _
renderstate **As Long)**

object
Object expression that resolves to a **Direct3DDevice3** object.

state
Device state variable that is being modified. This parameter can be any of the constants of the **CONST_D3DRENDERSTATETYPE** enumeration.

renderstate
New value for the Direct3DDevice render state. The meaning of this parameter is dependent on the value specified for *state*. For example, if *state* were D3DRENDERSTATE_SHADEMODE, the second parameter would be one of the constants of the **CONST_D3DSHADEMODE** enumeration.

Error Codes

If the method fails, it sets **Err.Number** to an error code and raises an error. The error code is DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

Applications should use the **Direct3DDevice3.SetTextureStageState** method to set texture states in favor of the legacy texture-related render states. For more information, see About Render States.

[#] IDH__dx_Direct3DDevice3.SetRenderState_d3d_vb

See Also

Direct3DDevice3.GetRenderState, **Direct3DDevice3.SetLightState**, **Direct3DDevice3.SetTransform**

Direct3DDevice3.SetRenderTarget

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.SetRenderTarget** method permits the application to easily route rendering output to a new DirectDraw surface as a render target.

object.**SetRenderTarget(***surface* **As DirectDrawSurface4)**

object
Object expression that resolves to a **Direct3DDevice3** object.

surface
A **DirectDrawSurface4** object for the previously created surface object that will be the new rendering target.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set one of the following values:

DDERR_INVALIDPARAMS	One of the arguments is invalid.
DDERR_INVALIDSURFACETYPE	The surface passed as the first parameter is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

When you change the rendering target, all of the handles associated with the previous rendering target become invalid. This means that you will have to reacquire all of the texture handles. If you are using ramp mode, you should also update the texture handles inside materials, by calling the **Direct3DMaterial3.SetMaterial** method. The **Direct3DDevice3.SetRenderTarget** method is most useful to applications that use the DrawPrimitive methods, especially when these applications do not use ramp mode.

If the new render target surface has different dimensions from the old (length, width, pixel-format), this method marks the viewport as invalid. The viewport may be revalidated after calling **Direct3DDevice3.SetRenderTarget** by calling **Direct3DViewport3.SetViewport** to restate viewport parameters that are compatible with the new surface.

[#] IDH__dx_Direct3DDevice3.SetRenderTarget_d3d_vb

Capabilities do not change with changes in the properties of the render target surface. Both the Direct3D HAL and the software rasterizers have only one opportunity to expose capabilities to the application. The system cannot expose different sets of capabilities depending on the format of the destination surface.

If a depth-buffer is attached to the new render target, it replaces the previous z-buffer for the context. Otherwise, the old z-buffer is detached and z-buffering is disabled.

If more than one depth-buffer is attached to the render target, this function fails.

See Also

Direct3DDevice3.GetRenderTarget

Direct3DDevice3.SetTexture

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.SetTexture** method assigns a texture to a given stage for a device.

object.**SetTexture(** _
stage **As Long**, _
texture **As Direct3DTexture2)**

object
Object expression that resolves to a **Direct3DDevice3** object.

stage
Stage identifier to which the texture will be set. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value allowed for *stage* is 7.

texture
A **Direct3DTexture2** object for the texture being set.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_Direct3DDevice3.SetTexture_d3d_vb

Remarks

Software devices do not support assigning a texture to more than one texture stage at a time.

See Also

Direct3DDevice3.GetTexture, **Direct3DDevice3.GetTextureStageState**, **Direct3DDevice3.SetTextureStageState**

Direct3DDevice3.SetTextureStageState

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.SetTextureStageState** method sets the state value for a currently assigned texture.

```
object.SetTextureStageState( _
    stage As Long, _
    state As CONST_D3DTEXTURESTAGESTATETYPE, _
    value As Long)
```

object
Object expression that resolves to a **Direct3DDevice3** object.

stage
Stage identifier of the texture for which the state value will be set. Stage identifiers are zero-based. Currently, devices can have up to 8 set textures, so the maximum allowable value allowed for *stage* is 7.

state
Texture state to be set. This parameter can be any constant of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

value
State value to be set. The meaning of this value is determined by the *state* parameter.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_Direct3DDevice3.SetTextureStageState_d3d_vb

Remarks

Applications should use this method to set texture states in favor of the legacy texture-related render states.

See Also

Direct3DDevice3.GetTextureStageState, **Direct3DDevice3.GetTexture**, **Direct3DDevice3.SetTexture**, Textures

Direct3DDevice3.SetTransform

[This is preliminary documentation and subject to change.]

The **Direct3DDevice3.SetTransform** method sets a single Direct3DDevice transformation-related state.

object.**SetTransform(_**
transformType **As CONST_D3DTRANSFORMSTATETYPE**, **_**
matrix **As D3DMATRIX)**

object
Object expression that resolves to a **Direct3DDevice3** object.

transformType
Device state variable that is being modified. This parameter can be any of the constants of the **CONST_D3DTRANSFORMSTATETYPE** enumeration.

matrix
A **D3DMATRIX** type that modifies the current transformation.

Error Codes

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice3.GetTransform, **Direct3DDevice3.SetLightState**, **Direct3DDevice3.SetRenderState**

Direct3DDevice3.ValidateDevice

[This is preliminary documentation and subject to change.]

IDH__dx_Direct3DDevice3.SetTransform_d3d_vb
IDH__dx_Direct3DDevice3.ValidateDevice_d3d_vb

The **Direct3DDevice3.ValidateDevice** method reports the device's ability to render the currently set texture blending operations and arguments in a single pass.

object.**ValidateDevice() As Long**

object
Object expression that resolves to a **Direct3DDevice3** object.

Return Values

If the method succeeds, the return value is the number of rendering passes to complete the desired effect through multipass rendering.

Error Codes

If the method fails, an error is raised and **Err.Number** may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
D3DERR_CONFLICTINGTEXTUREFILTER
D3DERR_TOOMANYOPERATIONS
D3DERR_UNSUPPORTEDALPHAARG
D3DERR_UNSUPPORTEDALPHAOPERATION
D3DERR_UNSUPPORTEDCOLORARG
D3DERR_UNSUPPORTEDCOLOROPERATION
D3DERR_UNSUPPORTEDFACTORVALUE
D3DERR_UNSUPPORTEDTEXTUREFILTER
D3DERR_WRONGTEXTUREFORMAT

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Current hardware does not necessarily implement all possible combinations of operations and arguments. You can determine whether a particular blending operation can be performed with given arguments by setting-up the desired blending operation, then calling the **ValidateDevice** method.

The **ValidateDevice** method uses the currently set render states, textures, and, texture stage states to perform validation at the time of the call. Any changes to these factors after the call invalidate the previous result, and the method must be called again before rendering a scene.

Using diffuse iterated values, either as an argument or as an operation (D3DTA_DIFFUSE or **D3DTOP_BLENDDIFFUSEALPHA**) is sparsely supported

on current hardware. Most hardware can only introduce iterated color data at the last texture operation stage.

Try to specify the texture (D3DTA_TEXTURE) for each stage as the first argument, in preference to the second argument.

Many cards do not support use of diffuse or scalar values at arbitrary texture stages. Often, these are only available at the first or last texture blending stage.

Many cards do not actually have a blending unit associated with the first texture that is capable of more than replicating alpha to color channels, or inverting the input. As a result, your application might need to use only the second texture stage if possible. On such hardware, the first unit is presumed to be in its default state, which has the first color argument set to D3DTA_TEXTURE with the **D3DTOP_SELECTARG1** operation.

Operations on the output alpha that are more intricate than or substantially different from the color operations are less likely to be supported.

Some hardware does not support simultaneous use of both D3DTA_TFACTOR and D3DTA_DIFFUSE.

Many cards do not support simultaneous use of multiple textures and mipmapped trilinear filtering. If trilinear filtering has been requested for a texture involved in multi-texture blending operations and validation fails, turn off trilinear filtering and revalidate. In this case, it might be best to perform multipass rendering instead.

See Also

Direct3DDevice3.GetTextureStageState, **Direct3DDevice3.SetTextureStageState**

Direct3DDevice3.Vertex

[#] [This is preliminary documentation and subject to change.]

The **Direct3DDevice3.Vertex** method adds a new Direct3D vertex to the primitive sequence started with a previous call to the **Direct3DDevice3.Begin** method.

object.**Vertex(***Vertex* **As Any)**

object
Object expression that resolves to a **Direct3DDevice3** object.

Vertex
Pointer to the next Direct3D vertex to be added to the currently started primitive sequence. This can be any of the Direct3D vertex types (**D3DLVERTEX**, **D3DTLVERTEX**, or **D3DVERTEX**) or a vertex specified in flexible vertex format. The vertex format must match the description specified in the preceding call to **Direct3DDevice3.Begin**.

[#] IDH__dx_Direct3DDevice3.Vertex_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DDevice3.Begin, **Direct3DDevice3.End**

Direct3DEnumDevices

[This is preliminary documentation and subject to change.]

Applications use the methods of the **Direct3DEnumDevices** class to retrieve information about the Direct3D devices present on a system. The **Direct3DEnumDevices** class is obtained by calling the **Direct3D3.GetDevicesEnum** method.

The methods of the **Direct3DEnumDevices** class can be organized into the following groups:

Device count	**GetCount**
Device information	**GetDescription**
	GetGuid
	GetHELDesc
	GetHWDesc
	GetName

Direct3DEnumDevices.GetCount

[This is preliminary documentation and subject to change.]

The **Direct3DEnumDevices.GetCount** method returns the number of Direct3D devices installed on the system.

object.**GetCount() As Long**

IDH__dx_Direct3DEnumDevices_d3d_vb
IDH__dx_Direct3DEnumDevices.GetCount_d3d_vb

object
Object expression that resolves to a **Direct3DEnumDevices** object.

Return Values

If the method succeeds, the number of Direct3D devices installed on the system is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DEnumDevices.GetDescription

[This is preliminary documentation and subject to change.]

The **Direct3DEnumDevices.GetDescription** method returns user-friendly description of the desired Direct3D device.

object.**GetDescription(***index* **As Long) As String**

object
Object expression that resolves to a **Direct3DEnumDevices** object.

index
Index of the enumerated device. This value can be between 1 and the value returned by the **Direct3DEnumDevices.GetCount** method.

Return Values

If the method succeeds, the return value is the user-friendly description of the enumerated device.

Error Codes

If the method fails, the method sets **Err.Number** to an error code and raises an error. For a list of possible error codes, see Direct3D Immediate Mode Error Codes.

Direct3DEnumDevices.GetGuid

[This is preliminary documentation and subject to change.]

The **Direct3DEnumDevices.GetDescription** method returns the globally-unique ID (GUID) of the desired Direct3D device.

IDH__dx_Direct3DEnumDevices.GetDescription_d3d_vb
IDH__dx_Direct3DEnumDevices.GetGuid_d3d_vb

object.**GetGuid(***index* **As Long) As String**

object
Object expression that resolves to a **Direct3DEnumDevices** object.

index
Index of the enumerated device. This value can be between 1 and the value returned by the **Direct3DEnumDevices.GetCount** method.

Return Values

If the method succeeds, the return value is the globally-unique identifier of the enumerated device. This value is used to create the device with a subsequent call to the **Direct3D3.CreateDevice** method.

Error Codes

If the method fails, the method sets **Err.Number** to an error code and raises an error. For a list of possible error codes, see Direct3D Immediate Mode Error Codes.

Direct3DEnumDevices.GetHELDesc

[#] [This is preliminary documentation and subject to change.]

The **Direct3DEnumDevices.GetHELDesc** method returns the emulated capabilities of the desired Direct3D device.

object.**GetHELDesc(***index* **As Long,** *helDesc* **As D3DDEVICEDESC)**

object
Object expression that resolves to a **Direct3DEnumDevices** object.

index
Index of the enumerated device. This value can be between 1 and the value returned by the **Direct3DEnumDevices.GetCount** method.

helDesc
A **D3DDEVICEDESC** type that contains the emulated capabilities of the Direct3D device.

Error Codes

If the method fails, an error is generated. For a list of possible error codes, see Direct3D Immediate Mode Error Codes.

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_Direct3DEnumDevices.GetHELDesc_d3d_vb

Direct3DEnumDevices.GetHWDesc

[This is preliminary documentation and subject to change.]

The **Direct3DEnumDevices.GetHWDesc** method returns the hardware capabilities of the desired Direct3D device.

object.**GetHWDesc(***index* **As Long,** *hwDesc* **As D3DDEVICEDESC)**

object
Object expression that resolves to a **Direct3DEnumDevices** object.

index
On a system with multiple Direct3D devices, this parameter represents the specific device.

hwDesc
A **D3DDEVICEDESC** type that contains the hardware capabilities of the Direct3D device.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DEnumDevices.GetName

[This is preliminary documentation and subject to change.]

The **Direct3DEnumDevices.GetName** method returns the user-friendly name of the desired Direct3D device.

object.**GetName(***index* **As Long) As String**

object
Object expression that resolves to a **Direct3DEnumDevices** object.

index
Index of the enumerated device. This value can be between 1 and the value returned by the **Direct3DEnumDevices.GetCount** method.

Return Values

If the method succeeds, the return value is the user-friendly name of the device.

Error Codes

If the method fails, the method sets **Err.Number** to an error code and raises an error. For a list of possible error codes, see Direct3D Immediate Mode Error Codes.

IDH__dx_Direct3DEnumDevices.GetHWDesc_d3d_vb
IDH__dx_Direct3DEnumDevices.GetName_d3d_vb

Direct3DEnumPixelFormats

[This is preliminary documentation and subject to change.]

Applications use the methods of the **Direct3DEnumPixelFormats** class to retrieve information about the pixel formats supported by a rendering device. The **Direct3D3EnumDevices** class is obtained by calling the **Direct3DDevice3.GetTextureFormatsEnum** and **Direct3D3.GetEnumZBufferFormats** methods.

This class consists of two methods.

- **GetCount**
- **GetItem**

Direct3DEnumPixelFormats.GetCount

[This is preliminary documentation and subject to change.]

The **Direct3DEnumPixelFormats.GetCount** method returns the number of supported pixel formats of the Direct3D device.

object.**GetCount() As Long**

object
Object expression that resolves to a **Direct3DEnumPixelFormats** object.

Return Values

If the method succeeds, the number of supported pixel formats is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DEnumPixelFormats.GetItem

[This is preliminary documentation and subject to change.]

The **Direct3DEnumPixelFormats.GetItem** method returns the description of the specified pixel format.

object.**GetItem(***index* **As Long,** *pixelFormat* **As DDPIXELFORMAT)**

object

IDH__dx_Direct3DEnumPixelFormats_d3d_vb
IDH__dx_Direct3DEnumPixelFormats.GetCount_d3d_vb
IDH__dx_Direct3DEnumPixelFormats.GetItem_d3d_vb

Object expression that resolves to a **Direct3DEnumPixelFormats** object.

index
The specific pixel format that you want a description for.

pixelFormat
A **DDPIXELFORMAT** type that describes the enumerated pixel format.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DLight

[This is preliminary documentation and subject to change.]

Applications use the methods of the **Direct3DLight** class to retrieve and set the capabilities of lights. This section is a reference to the methods of this class. For a conceptual overview, see Lights.

The **Direct3DLight** object is obtained by calling the **Direct3D3.CreateLight** method.

The methods of the **Direct3DLight** class can be organized into the following groups:

Get and set	**GetLight**
	SetLight

Direct3DLight.GetLight

[This is preliminary documentation and subject to change.]

The **Direct3DLight.GetLight** method retrieves the light information for the Direct3DLight object.

object.**GetLight(***light* **As D3DLIGHT2)**

object
Object expression that resolves to a **Direct3DLight** object.

light
A **D3DLIGHT2** type that will be filled with the current light data.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

IDH__dx_Direct3DLight_d3d_vb
IDH__dx_Direct3DLight.GetLight_d3d_vb

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DLight.SetLight

Direct3DLight.SetLight

[This is preliminary documentation and subject to change.]

The **Direct3DLight.SetLight** method sets the light information for the Direct3DLight object.

object.**SetLight(***light* **As D3DLIGHT2)**

object
Object expression that resolves to a **Direct3DLight** object.

light
A **D3DLIGHT2** type that will be filled with the current light data.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DLight.GetLight

Direct3DMaterial3

[This is preliminary documentation and subject to change.]

Applications use the methods of the **Direct3DMaterial3** class to retrieve and set the properties of materials. This section is a reference to the methods of this class. For a conceptual overview, see Materials.

You create this object by calling the **Direct3D3.CreateMaterial** method.

IDH__dx_Direct3DLight.SetLight_d3d_vb
IDH__dx_Direct3DMaterial3_d3d_vb

The methods of the **Direct3DMaterial3** class can be organized into the following groups:

Handles	**GetHandle**
Materials	**GetMaterial**
	SetMaterial

See Also

Materials, Lighting and Materials

Direct3DMaterial3.GetHandle

[This is preliminary documentation and subject to change.]

The **Direct3DMaterial3.GetHandle** method binds a material to a device, retrieving a handle that represents the association between the two. This handle is used in all Direct3D methods in which a material is to be referenced. A material can be used by only one device at a time.

If the device is destroyed, the material is disassociated from the device.

object.**GetHandle(***dev* **As Direct3DDevice3) As Long**

object
Object expression that resolves to a **Direct3DMaterial3** object.

dev
A **Direct3DDevice3** object for the rendering device to which the material is being bound.

Return Values

If the method succeeds, the return value is the material handle corresponding to the **Direct3DMaterial3** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to DDERR_INVALIDOBJECT.

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Retrieving Material Handles

IDH__dx_Direct3DMaterial3.GetHandle_d3d_vb

Direct3DMaterial3.GetMaterial

[#] [This is preliminary documentation and subject to change.]

The **Direct3DMaterial3.GetMaterial** method retrieves the material data for the Direct3DMaterial object.

object.**GetMaterial(***mat* **As D3DMATERIAL)**

object
Object expression that resolves to a **Direct3DMaterial3** object.

mat
A **D3DMATERIAL** type that will be filled with the current material properties.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DMaterial3.SetMaterial

Direct3DMaterial3.SetMaterial

[#] [This is preliminary documentation and subject to change.]

The **Direct3DMaterial3.SetMaterial** method sets the material data for the Direct3DMaterial object.

object.**SetMaterial(***mat* **As D3DMATERIAL)**

object
Object expression that resolves to a **Direct3DMaterial3** object.

mat
A **D3DMATERIAL** type that contains the material properties.

[#] IDH__dx_Direct3DMaterial3.GetMaterial_d3d_vb
[#] IDH__dx_Direct3DMaterial3.SetMaterial_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DMaterial3.GetMaterial, Retrieving Material Properties, Setting Material Properties

Direct3DTexture2

[This is preliminary documentation and subject to change.]

Applications use the methods of the **Direct3DTexture2** class to retrieve and set the properties of textures. This section is a reference to the methods of this class. For a conceptual overview, see Textures.

You create the **Direct3DTexture2** object by calling the **DirectDrawSurface4.GetTexture**

The methods of the **Direct3DTexture2** class can be organized into the following groups:

Handles	**GetHandle**
Loading	**Load**
Creating objects	**GetSurface**
Palette information	**PaletteChanged**

See Also

Textures

Direct3DTexture2.GetHandle

[This is preliminary documentation and subject to change.]

IDH__dx_Direct3DTexture2_d3d_vb
IDH__dx_Direct3DTexture2.GetHandle_d3d_vb

The **Direct3DTexture2.GetHandle** method obtains the texture handle to be used when rendering with **Direct3DDevice2** objects.

object.**GetHandle(***dev* **As Direct3DDevice3) As Long**

object
Object expression that resolves to a **Direct3DTexture2** object.

dev
A **Direct3DDevice3** object into which the texture is to be loaded.

Return Values

If the method succeeds, the return value is the texture handle corresponding to the **Direct3DTexture2** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

In the **Direct3DTexture** class, this method uses a pointer to a **Direct3DDevice** object instead of a **Direct3DDevice2** object.

Texture handles are used only device objects earlier than **Direct3DDevice3**. The **Direct3DDevice3** object references textures using texture object pointers, set through the **Direct3DDevice3.SetTexture** method.

Direct3DTexture2.GetSurface

[#] [This is preliminary documentation and subject to change.]

The **Direct3DTexture2.GetSurface** method creates a **DirectDrawSurface4** object.

object.**GetSurface() As DirectDrawSurface4**

object
Object expression that resolves to a **Direct3DTexture2** object.

[#] IDH__dx_Direct3DTexture2.GetSurface_d3d_vb

Return Values

If the method succeeds, a **DirectDrawSurface4** object is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set. For a list of possible error codes, see Direct3D Immediate Mode Error Codes.

Direct3DTexture2.Load

[This is preliminary documentation and subject to change.]

The **Direct3DTexture2.Load** method loads a system-memory texture surface into a video-memory texture surface. This method can be used to load texture mipmap chains (see remarks).

object.**Load(***tex* **As Direct3DTexture2)**

object
Object expression that resolves to a **Direct3DTexture2** object.

tex
The texture to load.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set. For a list of possible return values, see Direct3D Immediate Mode Error Codes.

Remarks

This method uses hardware-accelerated blit operations to load data from the source texture into the destination texture.

If both textures are mipmaps, the method will copy the mipmap levels from the source mipmap that match those of the destination mipmap. If the destination mipmap uses levels-of-detail not present in the source mipmap, the method fails.

Direct3DTexture2.PaletteChanged

[This is preliminary documentation and subject to change.]

The **Direct3DTexture2.PaletteChanged** method informs the driver that the palette has changed on a texture surface.

object.**PaletteChanged(***start* **As Long**, *count* **As Long)**

IDH__dx_Direct3DTexture2.Load_d3d_vb
IDH__dx_Direct3DTexture2.PaletteChanged_d3d_vb

object
Object expression that resolves to a **Direct3DTexture2** object.

start
Index of first palette entry that has changed.

count
Number of palette entries that have changed.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

This method is particularly useful for applications that play video clips and therefore require palette-changing capabilities.

This method only affects the legacy ramp device. For all other devices, this method takes no action and returns D3D_OK.

Direct3DVertexBuffer

[#] [This is preliminary documentation and subject to change.]

Applications use the methods of the **Direct3DVertexBuffer** class to manipulate a collection of vertices for use with the **Direct3DDevice3.DrawPrimitiveVB** and **Direct3DDevice3.DrawIndexedPrimitiveVB** rendering methods. This section is a reference to the methods of this class. For a conceptual overview, see Vertex Buffers.

This methods of the **Direct3DVertexBuffer** class can be organized into the following groups:

Information	**GetVertexBufferDesc**
Vertex data	**GetVertices**
	Lock
	Optimize
	ProcessVertices
	SetVertices
	Unlock

See Also

[#] IDH__dx_Direct3DVertexBuffer_d3d_vb

Vertex Buffers

Direct3DVertexBuffer.GetVertexBufferDesc

[This is preliminary documentation and subject to change.]

The **Direct3DVertexBuffer.GetVertexBufferDesc** method retrieves a description of the vertex buffer.

object.**GetVertexBufferDesc(***desc* **As D3DVERTEXBUFFERDESC)**

object
Object expression that resolves to a **Direct3DVertexBuffer** object.

desc
A **D3DVERTEXBUFFERDESC** type that will be filled with a description of the vertex buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be DDERR_INVALIDPARAMS or another error code.

Direct3DVertexBuffer.GetVertices

[This is preliminary documentation and subject to change.]

The **Direct3DVertexBuffer.GetVertices** method returns the vertices in the vertex buffer.

object.**GetVertices(***startIndex* **As Long, _**
count **As Long, _**
verts **As Any)**

object
Object expression that resolves to a **Direct3DVertexBuffer** object.

startIndex

count

verts

IDH__dx_Direct3DVertexBuffer.GetVertexBufferDesc_d3d_vb
IDH__dx_Direct3DVertexBuffer.GetVertices_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DVertexBuffer.Lock

[#] [This is preliminary documentation and subject to change.]

The **Direct3DVertexBuffer.Lock** methods locks a vertex buffer and obtains a pointer to the vertex buffer memory.

object.**Lock(***flags* **As CONST_DDLOCKFLAGS)**

object
Object expression that resolves to a **Direct3DVertexBuffer** object.

flags
One of the constants of the **CONST_DDLOCKFLAGS** enumeration indicating how the vertex buffer memory should be locked.

DDLOCK_EVENT
This flag is not currently implemented.

DDLOCK_NOSYSLOCK
If possible, do not take the Win16Mutex (also known as Win16Lock).

DDLOCK_READONLY
Indicates that the memory being locked will only be read from.

DDLOCK_SURFACEMEMORYPTR
Indicates that a valid memory pointer to the vertex buffer should be returned; this is the default.

DDLOCK_WAIT
If a lock cannot be obtained immediately, the method retries until a lock is obtained or another error occurs.

DDLOCK_WRITEONLY
Indicates that the memory being locked will only be written to.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_VERTEXBUFFEROPTIMIZED

DDERR_INVALIDPARAMS

DDERR_OUTOFMEMORY

DDERR_SURFACEBUSY

DDERR_SURFACELOST

[#] IDH__dx_Direct3DVertexBuffer.Lock_d3d_vb

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

After locking the vertex buffer, you can access the memory until a corresponding call to **Direct3DVertexBuffer.Unlock**.

You cannot render from a locked vertex buffer; calls to the **Direct3DDevice3.DrawIndexedPrimitiveVB** or **Direct3DDevice3.DrawPrimitiveVB** method using a locked buffer will fail, returning D3DERR_VERTEXBUFFERLOCKED.

This method often causes the system to hold the Win16Mutex until you call the **Direct3DVertexBuffer.Unlock** method. GUI debuggers cannot operate while the Win16Mutex is held.

See Also

Direct3DVertexBuffer.Unlock

Direct3DVertexBuffer.Optimize

[#] [This is preliminary documentation and subject to change.]

The **Direct3DVertexBuffer.Optimize** method converts an unoptimized vertex buffer into an optimized vertex buffer.

object.**Optimize(***dev* **As Direct3DDevice3)**

object
Object expression that resolves to a **Direct3DVertexBuffer** object.

dev
A **Direct3DDevice3** object of the device for which this vertex buffer will be optimized.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_VERTEXBUFFEROPTIMIZED
D3DERR_VERTEXBUFFERLOCKED
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_Direct3DVertexBuffer.Optimize_d3d_vb

See Also

Optimizing a Vertex Buffer, Vertex Buffers

Direct3DVertexBuffer.ProcessVertices

[#] [This is preliminary documentation and subject to change.]

The **Direct3DVertexBuffer.ProcessVertices** method processes untransformed vertices into a transformed or optimized vertex buffer.

object.**ProcessVertices(** _
vertexOp **As CONST_D3DVOPFLAGS**, _
destIndex **As Long**, _
count **As Long**, _
srcBuffer **As Direct3DVertexBuffer**, _
srcIndex **As Long**, _
dev **As Direct3DDevice3)**

object
Object expression that resolves to a **Direct3DVertexBuffer** object.

vertexOp
Flags defining how the method processes the vertices as they are transferred from the source buffer. You can specify any combination of the following constants of the **CONST_D3DVOPFLAGS** enumeration:

D3DVOP_CLIP
Transform the vertices and clip any vertices that exist outside the viewing frustum. This flag cannot be used with vertex buffers that do not contain clipping information (for example, created with the D3DDP_DONOTCLIP flag).

D3DVOP_EXTENTS
Transform the vertices, then update the extents of the screen rectangle when the vertices are rendered. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice3.GetClipStatus** will not have been updated to account for the vertices when they are rendered.

D3DVOP_LIGHT
Light the vertices.

D3DVOP_TRANSFORM
Transform the vertices using the world, view, and projection matrices. This flag must always be set.

destIndex
Index into the destination vertex buffer (this buffer) where the vertices will be placed after processing.

count
Number of vertices in the source buffer to process.

[#] IDH__dx_Direct3DVertexBuffer.ProcessVertices_d3d_vb

srcBuffer
A **Direct3DVertexBuffer** object for the source vertex buffer.

srcIndex
Index of the first vertex in the source buffer to be processed.

dev
A **Direct3DDevice3** object for the device that will be used to transform the vertices.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_INVALIDVERTEXFORMAT
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACEBUSY
DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

You must always include the D3DVOP_TRANSFORMED flag in the *vertexOp* parameter. If you fail to include this flag, the method will fail, returning DDERR_INVALIDPARAMS.

See Also

Processing Vertices, Vertex Buffers

Direct3DVertexBuffer.SetVertices

[This is preliminary documentation and subject to change.]

The **Direct3DVertexBuffer.SetVertices** method sets the vertices in the vertex buffer.

```
object.SetVertices(startIndex As Long, _
    count As Long, _
    verts As Any)
```

object
Object expression that resolves to a **Direct3DVertexBuffer** object.

startIndex

IDH__dx_Direct3DVertexBuffer.SetVertices_d3d_vb

count

verts

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Direct3DVertexBuffer.Unlock

[This is preliminary documentation and subject to change.]

The **Direct3DVertexBuffer.Unlock** method unlocks a previously locked vertex buffer.

object.**Unlock()**

object
Object expression that resolves to a **Direct3DVertexBuffer** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACEBUSY
DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DVertexBuffer.Lock, Accessing Vertex Buffer Memory

Direct3DViewport3

[This is preliminary documentation and subject to change.]

IDH__dx_Direct3DVertexBuffer.Unlock_d3d_vb
IDH__dx_Direct3DViewport3_d3d_vb

Applications use the methods of the **Direct3DViewport3** class to retrieve and set the properties of viewports. This section is a reference to the methods of this class. For a conceptual overview, see Viewports and Clipping.

The **Direct3DViewport3** class offers the same services as the **Direct3DViewport2** class, but adds the **Clear2** method, which simultaneously clears the viewport, depth-buffer, and stencil buffer.

You create the **Direct3DViewport3** object by calling the **Direct3D3.CreateViewport** method.

The methods of the **Direct3DViewport3** class can be organized into the following groups:

Backgrounds	**GetBackground** **GetBackgroundDepth** **SetBackground** **SetBackgroundDepth**
Lights	**AddLight** **DeleteLight** **LightElements** **NextLight**
Materials and viewports	**Clear** **Clear2** **GetViewport** **GetViewport2** **SetViewport** **SetViewport2**
Transformation	**TransformVertices**

See Also

Viewports and Clipping

Direct3DViewport3.AddLight

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_Direct3DViewport3.AddLight_d3d_vb

The **Direct3DViewport3.AddLight** method adds the specified light to the list of Direct3DLight objects associated with this viewport and increments the reference count of the light object.

object.**AddLight(***light* **As Direct3DLight)**

object
Object expression that resolves to a **Direct3DViewport3** object.

light
A **Direct3DLight** object that should be associated with this **Direct3DViewport3** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Direct3DViewport3.Clear

[#] [This is preliminary documentation and subject to change.]

The **Direct3DViewport3.Clear** method clears the viewport or a set of rectangles in the viewport to the current background material.

object.**Clear(***count* **As Long**, _
recs() **As D3DRECT**, _
flags **As CONST_D3DCLEARFLAGS)**

object
Object expression that resolves to a **Direct3DViewport3** object.

count
Number of rectangles in the *recs()* array.

recs()
An array of **D3DRECT** types. Each rectangle uses screen coordinates that correspond to points on the render target surface. Coordinates are clipped to the bounds of the viewport rectangle.

flags
One or both of the following constants of the **CONST_D3DCLEARFLAGS** enumeration indicating what to clear: the rendering target, the depth-buffer, or both.

D3DCLEAR_TARGET
Clear the rendering target to the background material (if set).

[#] IDH__dx_Direct3DViewport3.Clear_d3d_vb

D3DCLEAR_ZBUFFER
Clear the depth-buffer or set it to the current background depth field (if set).

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_VIEWPORTHASNODEVICE
D3DERR_ZBUFFER_NOTPRESENT
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method does not support clearing the stencil buffer. To clear the stencil buffer, use the **Direct3DViewport3.Clear2** method.

See Also

Direct3DViewport3.Clear2

Direct3DViewport3.Clear2

[This is preliminary documentation and subject to change.]

The **Direct3DViewport3.Clear2** method clears the viewport (or a set of rectangles in the viewport) to a specified RGBA color, clears the depth-buffer, and erases the stencil buffer.

```
object.Clear2(count As Long, _
   recs() As D3DRECT, _
   flags As CONST_D3DCLEARFLAGS, _
   color As Long, _
   z As Single, _
   stencil As Long)
```

object
Object expression that resolves to a **Direct3DViewport3** object.

count
Number of rectangles in the array at *recs()*.

recs()

IDH__dx_Direct3DViewport3.Clear2_d3d_vb

An array of **D3DRECT** types that describe the rectangles to be cleared. Set a rectangle to the dimensions of the rendering target to clear the entire surface. Each rectangle uses screen coordinates that correspond to points on the render target surface. Coordinates are clipped to the bounds of the viewport rectangle.

flags

Flags indicating which surfaces should be cleared. This parameter can be any combination of the following constants of the **CONST_D3DCLEARFLAGS** enumeration, but at least one constant must be used:

D3DCLEAR_TARGET
Clear the rendering target to the color in the *dwColor* parameter.

D3DCLEAR_STENCIL
Clear the stencil buffer to the value in the *dwStencil* parameter.

D3DCLEAR_ZBUFFER
Clear the depth-buffer to the value in the *dvZ* parameter.

color

32-bit RGBA color value to which the render target surface will be cleared.

z

New z value that this method stores in the depth-buffer. This parameter can range from 0.0 to 1.0, inclusive. The value of 0.0 represents the nearest distance to the viewer, and 1.0 the farthest distance.

stencil

Integer value to store in each stencil buffer entry. This parameter can range from 0 to 2^n-1 inclusive, where *n* is the bit depth of the stencil buffer.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_STENCILBUFFER_NOTPRESENT

D3DERR_VIEWPORTHASNODEVICE

D3DERR_ZBUFFER_NOTPRESENT

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

This method fails if you specify the D3DCLEAR_ZBUFFER or D3DCLEAR_STENCIL flags when the render target does not have an attached depth-buffer. This behavior differs from the **Direct3DViewport3.Clear** method, which will succeed if under these circumstances.

If you specify the D3DCLEAR_STENCIL flag when the depth-buffer format doesn't contain stencil buffer information, this method fails.

This method ignores the current background material for the viewport; to clear a viewport using the background material, use the **Direct3DViewport3.Clear** method.

See Also

Direct3DViewport3.Clear

Direct3DViewport3.DeleteLight

[This is preliminary documentation and subject to change.]

The **Direct3DViewport3.DeleteLight** method removes the specified light from the list of Direct3DLight objects associated with this viewport, and decrements the reference count for the light object.

object.**DeleteLight(***light* **As Direct3DLight)**

object
Object expression that resolves to a **Direct3DViewport3** object.

light
A **Direct3DLight** object that should be disassociated with this **Direct3DViewport3** object.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Direct3DViewport3.GetBackground

[This is preliminary documentation and subject to change.]

The **Direct3DViewport3.GetBackground** method retrieves the handle to a material that represents the current background associated with the viewport.

object.**GetBackground(***hdl* **As Long**, *stat* **As Long)**

object
Object expression that resolves to a **Direct3DViewport3** object.

hdl
The handle to the material being used as the background.

IDH__dx_Direct3DViewport3.DeleteLight_d3d_vb
IDH__dx_Direct3DViewport3.GetBackground_d3d_vb

stat
Indicator of whether a background is associated with the viewport. If this parameter is False, no background is associated with the viewport. This will be non zero for True.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DViewport3.SetBackground

Direct3DViewport3.GetBackgroundDepth

[#] [This is preliminary documentation and subject to change.]

The **Direct3DViewport3.GetBackgroundDepth** method retrieves a DirectDraw surface that represents the current background-depth field associated with the viewport.

object.**GetBackgroundDepth(***status* **As Long) _**
As DirectDrawSurface4

object
Object expression that resolves to a **Direct3DViewport3** object.

status
A parameter that is set to False if no background depth is associated with the viewport.

Return Values

If the method succeeds, the return value is a **DirectDrawSurface4** object representing the background depth.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

[#] IDH__dx_Direct3DViewport3.GetBackgroundDepth_d3d_vb

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DViewport3.SetBackgroundDepth

Direct3DViewport3.GetViewport

[This is preliminary documentation and subject to change.]

The **Direct3DViewport3.GetViewport** method retrieves the viewport registers of the viewport. This method is provided for backward compatibility. It has been superseded by the **Direct3DViewport3.GetViewport2** method.

object.**GetViewport(***vp* **As D3DVIEWPORT)**

object
Object expression that resolves to a **Direct3DViewport3** object.

vp
A **D3DVIEWPORT** type representing the viewport.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DViewport3.GetViewport2, **Direct3DViewport3.SetViewport**

Direct3DViewport3.GetViewport2

[This is preliminary documentation and subject to change.]

The **Direct3DViewport3.GetViewport2** method retrieves the viewport registers of the viewport.

object.**GetViewport2(***vp* **As D3DVIEWPORT2)**

IDH__dx_Direct3DViewport3.GetViewport_d3d_vb
IDH__dx_Direct3DViewport3.GetViewport2_d3d_vb

object
Object expression that resolves to a **Direct3DViewport3** object.

vp
A **D3DVIEWPORT2** type representing the viewport.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DViewport3.SetViewport2

Direct3DViewport3.LightElements

[#] [This is preliminary documentation and subject to change.]

The **Direct3DViewport3.LightElements** method is not currently implemented.

object.**LightElements(***elementCount* **As Long**, _
lightData() **As D3DLIGHTDATA)**

object
Object expression that resolves to a **Direct3DViewport3** object.

elementCount

lightData()

Direct3DViewport3.NextLight

[#] [This is preliminary documentation and subject to change.]

The **Direct3DViewport3.NextLight** method enumerates the **Direct3DLight** objects associated with the viewport.

object.**NextLight(***l1* **As Direct3DLight**, _
flags **As CONST_D3DNEXTFLAGS) As Direct3DLight**

[#] IDH__dx_Direct3DViewport3.LightElements_d3d_vb
[#] IDH__dx_Direct3DViewport3.NextLight_d3d_vb

object
Object expression that resolves to a **Direct3DViewport3** object.

l1
A **Direct3DLight** object representing the light in the list of lights associated with this viewport object.

flags
Flags specifying which light to retrieve from the list of lights. This must be set to one of the following constants of the **CONST_D3DNEXTFLAGS** enumeration:

D3DNEXT_HEAD
Retrieve the item at the beginning of the list.

D3DNEXT_NEXT
Retrieve the next item in the list.

D3DNEXT_TAIL
Retrieve the item at the end of the list.

Return Values

If the method succeeds, the return value a **Direct3DLight** object representing the requested light in the list of lights associated with this viewport object. The requested light is specified in the *flags* parameter.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Direct3DViewport3.SetBackground

[#] [This is preliminary documentation and subject to change.]

The **Direct3DViewport3.SetBackground** method sets the background material associated with the viewport.

object.**SetBackground(***hdl* **As Long)**

object
Object expression that resolves to a **Direct3DViewport3** object.

hdl
Material handle that will be used as the background.

[#] IDH__dx_Direct3DViewport3.SetBackground_d3d_vb

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

Direct3DViewport3.GetBackground

Direct3DViewport3.SetBackgroundDepth

[#] [This is preliminary documentation and subject to change.]

The **Direct3DViewport3.SetBackgroundDepth** method sets the background-depth field for the viewport.

object.**SetBackgroundDepth(***surface* **As DirectDrawSurface4)**

object
Object expression that resolves to a **Direct3DViewport3** object.

surface
A **DirectDrawSurface4** object representing the background depth.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

The depth-buffer is filled with the specified depth field when the **Direct3DViewport3.Clear** method is called and the D3DCLEAR_ZBUFFER flag is specified. The bit depth must be 16 bits.

[#] IDH__dx_Direct3DViewport3.SetBackgroundDepth_d3d_vb

See Also

Direct3DViewport3.GetBackgroundDepth

Direct3DViewport3.SetViewport

[This is preliminary documentation and subject to change.]

The **Direct3DViewport3.SetViewport** method sets the viewport registers of the viewport. This method is provided for backward compatibility. It has been superseded by the **Direct3DViewport3.SetViewport2** method.

object.**SetViewport(***vp* **As D3DVIEWPORT)**

object
Object expression that resolves to a **Direct3DViewport3** object.

vp
A **D3DVIEWPORT** type that contains the new viewport.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

You cannot set viewport parameters unless the viewport is associated with a rendering device (by calling the **Direct3DDevice3.AddViewport** method).

See Also

Direct3DViewport3.GetViewport, **Direct3DViewport3.SetViewport2**, Using Viewports, Viewports and Clipping

Direct3DViewport3.SetViewport2

[This is preliminary documentation and subject to change.]

The **Direct3DViewport3.SetViewport2** method sets the viewport registers of the viewport.

IDH__dx_Direct3DViewport3.SetViewport_d3d_vb
IDH__dx_Direct3DViewport3.SetViewport2_d3d_vb

object.**SetViewport2(***vp* **As D3DVIEWPORT2)**

object
Object expression that resolves to a **Direct3DViewport3** object.

vp
A **D3DVIEWPORT2** type that contains the new viewport parameters.

Error Codes

If the method fails, an error is raised and **Err.Number** may be set to one of the following values:

D3DERR_VIEWPORTHASNODEVICE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

You cannot set viewport parameters unless the viewport is associated with a rendering device (by calling the **Direct3DDevice3.AddViewport** method). For details, see Preparing to Use a Viewport.

The dvMinZ and dvMaxZ members of the associated D3DVIEWPORT2 type must not contain identical values.

See Also

Direct3DViewport3.GetViewport2, Using Viewports, Viewports and Clipping

Direct3DViewport3.TransformVertices

[#] [This is preliminary documentation and subject to change.]

The **Direct3DViewport3.TransformVertices** method transforms a set of vertices by the transformation matrix.

object.**TransformVertices(** _
vertexCount **As Long**, _
transformdata **As D3DTRANSFORMDATA**, _
InLVerts() As **D3DLVERTEX**, _
OutTLVerts() As **D3DTLVERTEX**, _
OutHVerts() As **D3DHVERTEX**, _
flags **As CONST_D3DTRANSFORMFLAGS,** _
offscreen **As Long)**

[#] IDH__dx_Direct3DViewport3.TransformVertices_d3d_vb

object
Object expression that resolves to a **Direct3DViewport3** object.

vertexCount
Number of vertices in the *InLVerts* parameter to be transformed.

transformdata
Variable of type **D3DTRANSFORMDATA** that contains the vertices to be transformed. See Remarks.

InLVerts()
Variable array of type **D3DLVERTEX** that contains the lit vertices to be transformed.

OutTLVerts()
Variable array of type **D3DTLVERTEX** that will contain the transformed vertices if the call succeeds.

OutHVerts()
Variable array of type **D3DHVERTEX** that will contain the homogeneous transformed vertices if the call succeeds.

flags
One of the following constants of the **CONST_D3DTRANSFORMFLAGS** enumeration. See the comments section following the parameter description for a discussion of how to use these flags.

D3DTRANSFORM_CLIPPED
Transform the vertices and adjust the rectangle in the **rExtent** member of the associated **D3DTRANSFORMDATA** type to reflect the new extents.

D3DTRANSFORM_UNCLIPPED
Transform the vertices without updating the extents.

offscreen
Variable that will be set to indicate the orientation of the transformed vertices. If this variable is non-zero after the call, all the transformed vertices are outside the viewing volume. If this is zero, then all or some of the vertices are visible.

Error Codes

If the method fails, it sets **Err.Number** to one of the following values and raises an error:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

If the *flags* parameter is set to D3DTRANSFORM_CLIPPED, this method uses the current transformation matrix to transform a set of vertices, checking the resulting vertices to see if they are within the viewing frustum. The homogeneous part of the **D3DLVERTEX** type within *data()* will be set if the vertex is clipped; otherwise only

the screen coordinates will be set. The clip intersection of all the vertices transformed is returned in *offscreen*. That is, if *offscreen* is nonzero, all the vertices were off-screen and not straddling the viewport.

Initialize the **rExtent** member of the **D3DTRANSFORMDATA** type to a **D3DRECT** type that describes a 2-D bounding rectangle (extents) that the method will "grow" if the transformed vertices do not fit within it. If the transformed vertices are outside the provided extents, the method adjusts the extents to fit the vertices, otherwise no changes are made. If the *flags* parameter is set to D3DTRANSFORM_UNCLIPPED, this method transforms the vertices assuming that the resulting coordinates will be within the viewing frustum. If clipping is requested by setting the *flags* parameter to D3DTRANSFORM_CLIPPED, the method adjusts the extents to fit only the transformed vertices that are within the viewing area.

The **lClip** member of **D3DTRANSFORMDATA** can help the transformation module determine whether the geometry will need clipping against the viewing volume. Before transforming a geometry, high-level software often can test whether bounding boxes or bounding spheres are wholly within the viewing volume, allowing clipping tests to be skipped, or wholly outside the viewing volume, allowing the geometry to be skipped entirely.

Types

[This is preliminary documentation and subject to change.]

This section contains information about the following types used with Direct3D Immediate Mode.

- **D3DCLIPSTATUS**
- **D3DCOLORVALUE**
- **D3DDEVICEDESC**
- **D3DFINDDEVICERESULT2**
- **D3DFINDDEVICESEARCH**
- **D3DHVERTEX**
- **D3DLIGHT2**
- **D3DLIGHTDATA**
- **D3DLIGHTINGCAPS**
- **D3DLVERTEX**
- **D3DMATERIAL**
- **D3DMATRIX**
- **D3DPRIMCAPS**
- **D3DRECT**
- **D3DSTATS**
- **D3DTLVERTEX**

- **D3DTRANSFORMDATA**
- **D3DVECTOR**
- **D3DVERTEX**
- **D3DVERTEXBUFFERDESC**
- **D3DVIEWPORT**
- **D3DVIEWPORT2**
- **DXDRIVERINFO**

D3DCLIPSTATUS

[#] [This is preliminary documentation and subject to change.]

The **D3DCLIPSTATUS** type describes the current clip status and extents of the clipping region.

```
Type D3DCLIPSTATUS
    lFlags As Long
    lStatus As Long
    maxx As Single
    maxy As Single
    maxz As Single
    minx As Single
    miny As Single
    minz As Single
End Type
```

lFlags

Flags describing whether this structure describes 2-D extents, 3-D extents, or the clip status. This member can be a combination of the following flags from the **CONST_D3DCLIPSTATUSFLAGS** enumeration:

D3DCLIPSTATUS_STATUS
The structure describes the current clip status.

D3DCLIPSTATUS_EXTENTS2
The structure describes the current 2-D extents. This flag cannot be combined with D3DCLIPSTATUS_EXTENTS3.

D3DCLIPSTATUS_EXTENTS3
Not currently implemented.

lStatus

Describes the current clip status. Clipping flags. This member can be one or more of the following constants of the **CONST_D3DCLIPFLAGS** enumeration:

Combination and General Flags

D3DSTATUS_CLIPINTERSECTIONALL

[#] IDH__dx_D3DCLIPSTATUS_d3d_vb

Combination of all CLIPINTERSECTION flags.

D3DSTATUS_CLIPUNIONALL
Combination of all CLIPUNION flags.

D3DSTATUS_DEFAULT
Combination of D3DSTATUS_CLIPINTERSECTIONALL and D3DSTATUS_ZNOTVISIBLE flags. This value is the default.

D3DSTATUS_ZNOTVISIBLE
Indicates that the rendered primitive is not visible. This flag is set or cleared by the system when rendering with z-checking enabled (see D3DRENDERSTATE_ZVISIBLE).

Clip Intersection Flags

D3DSTATUS_CLIPINTERSECTIONBACK
Logical **AND** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM
Logical **AND** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT
Logical **AND** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through
D3DSTATUS_CLIPINTERSECTIONGEN5
Logical **AND** of the clip flags for application-defined clipping planes.

D3DSTATUS_CLIPINTERSECTIONLEFT
Logical **AND** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT
Logical **AND** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP
Logical **AND** of the clip flags for the vertices compared to the top of the viewing frustum.

Clip Union Flags

D3DSTATUS_CLIPUNIONBACK
Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM
Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT
Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5
Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT
Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT
Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP
Equal to D3DCLIP_TOP.

Basic Clipping Flags

D3DCLIP_BACK
All vertices are clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM
All vertices are clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT
All vertices are clipped by the front plane of the viewing frustum.

D3DCLIP_LEFT
All vertices are clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT
All vertices are clipped by the right plane of the viewing frustum.

D3DCLIP_TOP
All vertices are clipped by the top plane of the viewing frustum.

D3DCLIP_GEN0 through D3DCLIP_GEN5
Application-defined clipping planes.

maxx, minx, maxy, miny, maxz, minz
The x, y, and z extents of the current clipping region.

See Also

Direct3DDevice3.GetClipStatus, **Direct3DDevice3.SetClipStatus**

D3DCOLORVALUE

[#] [This is preliminary documentation and subject to change.]

The **D3DCOLORVALUE** type describes color values for the **D3DLIGHT2** and **D3DMATERIAL** types.

```
Type D3DCOLORVALUE
   a As Single
   b As Single
   g As Single
   r As Single
End Type
```

a, **b**, **g** and **r**
Values specifying the red, green, blue, and alpha components of a color. These values generally range from 0 to 1, with 0 being black.

[#] IDH__dx_D3DCOLORVALUE_d3d_vb

Remarks

You can set the members of this type to values outside the range of 0 to 1 to implement some unusual effects. Values greater than 1 produce strong lights that tend to wash out a scene. Negative values produce dark lights, which actually remove light from a scene.

D3DDEVICEDESC

[#] [This is preliminary documentation and subject to change.]

The **D3DDEVICEDESC** type contains a description of the current device. This type is used to query the current device by such methods as **Direct3DDevice3.GetCaps**.

```
Type D3DDeviceDesc
    dlcLightingCaps As D3DLIGHTINGCAPS
    dpcLineCaps As D3DPRIMCAPS
    dpcTriCaps As D3DPRIMCAPS
    dvExtentsAdjust As Single
    dvGuardBandBottom As Single
    dvGuardBandLeft As Single
    dvGuardBandRight As Single
    dvGuardBandTop As Single
    lClipping As Long
    lColorModel As CONST_D3DCOLORMODEL
    lDevCaps As CONST_D3DDEVICEDESCCAPS
    lDeviceRenderBitDepth As Long
    lDeviceZBufferBitDepth As Long
    lFlags As CONST_D3DDEVICEDESCFLAGS
    lFVFCaps As CONST_D3DFVFCAPSFLAGS
    lMaxAnisotropy As Long
    lMaxBufferSize As Long
    lMinStippleHeight, lMaxStippleHeight As Long
    lMinStippleWidth, lMaxStippleWidth As Long
    lMaxTextureAspectRatio As Long
    lMaxTextureWidth, lMaxTextureHeight As Long
    lMaxTextureRepeat As Long
    lMaxVertexCount As Long
    lMinTextureWidth, lMinTextureHeight As Long
    lStencilCaps As CONST_D3DSTENCILCAPSFLAGS
    lTextureOpCaps As CONST_D3DTEXOPCAPSFLAGS
    lTransformCaps As CONST_D3DTRANSFORMCAPS
    nMaxSimultaneousTextures As Integer
    nMaxTextureBlendStages As Integer
End Type
```

[#] IDH__dx_D3DDEVICEDESC_d3d_vb

dlcLightingCaps
One of the members of the **D3DLIGHTINGCAPS** structure, specifying the lighting capabilities of the device.

dpcLineCaps and **dpcTriCaps**
D3DPRIMCAPS type defining the device's support for line-drawing and triangle primitives.

dvExtentsAdjust
Number of pixels to adjust the extents rectangle outward to accommodate antialiasing kernels.

dvGuardBandLeft, **dvGuardBandTop**, **dvGuardBandRight**, and **dvGuardBandBottom**
The screen-space coordinates of the guard-band clipping region. Coordinates inside this rectangle but outside the viewport rectangle will automatically be clipped.

lClipping
Non zero if the device can perform 3-D clipping.

lColorModel
One of the constants of the **CONST_D3DCOLORMODEL** enumeration, specifying the color model for the device.

lDevCaps
One of the following constants of the **CONST_D3DDEVICEDESCCAPS** enumeration identifying the capabilities of the device.

D3DDEVCAPS_CANRENDERAFTERFLIP
Device can queue rendering commands after a page flip. Applications should not change their behavior if this flag is set; this capability simply means that the device is relatively fast.

D3DDEVCAPS_DRAWPRIMTLVERTEX
Device exports a DrawPrimitive-aware HAL.

D3DDEVCAPS_EXECUTESYSTEMMEMORY
Device can use execute buffers from system memory.

D3DDEVCAPS_EXECUTEVIDEOMEMORY
Device can use execute buffer from video memory.

D3DDEVCAPS_FLOATTLVERTEX
Device accepts floating point for post-transform vertex data.

D3DDEVCAPS_SORTDECREASINGZ
Device needs data sorted for decreasing depth.

D3DDEVCAPS_SORTEXACT
Device needs data sorted exactly.

D3DDEVCAPS_SORTINCREASINGZ
Device needs data sorted for increasing depth.

D3DDEVCAPS_TEXREPEATNOTSCALEDBYSIZE
Device defers scaling of texture indices by the texture size until after the texture address mode is applied.

D3DDEVCAPS_TEXTURENONLOCALVIDMEM
Device can retrieve textures from non-local video (AGP) memory.

D3DDEVCAPS_TEXTURESYSTEMMEMORY
Device can retrieve textures from system memory.

D3DDEVCAPS_TEXTUREVIDEOMEMORY
Device can retrieve textures from device memory.

D3DDEVCAPS_TLVERTEXSYSTEMMEMORY
Device can use buffers from system memory for transformed and lit vertices.

D3DDEVCAPS_TLVERTEXVIDEOMEMORY
Device can use buffers from video memory for transformed and lit vertices.

lDeviceRenderBitDepth
Device's rendering bit-depth. This can be one or more of the following constants from the **CONST_DDBITDEPTHFLAGS** enumeration: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

lDeviceZBufferBitDepth
Device's depth-buffer bit-depth. This can be one or more of the following constants from the **CONST_DDBITDEPTHFLAGS** enumeration: DDBD_8, DDBD_16, DDBD_24, or DDBD_32.

lFlags
Constants of the **CONST_D3DDEVICEDESCFLAGS** enumeration identifying the members of this type that contain valid data.

D3DDD_BCLIPPING
The **lClipping** member is valid.

D3DDD_COLORMODEL
The **lColorModel** member is valid.

D3DDD_DEVCAPS
The **lDevCaps** member is valid.

D3DDD_DEVICERENDERBITDEPTH
The **lDeviceRenderBitDepth** member is valid.

D3DDD_DEVICEZBUFFERBITDEPTH
The **lDeviceZBufferBitDepth** member is valid.

D3DDD_LIGHTINGCAPS
The **dlcLightingCaps** member is valid.

D3DDD_LINECAPS
The **dpcLineCaps** member is valid.

D3DDD_MAXBUFFERSIZE
The **lMaxBufferSize** member is valid.

D3DDD_MAXVERTEXCOUNT
The **lMaxVertexCount** member is valid.

D3DDD_TRANSFORMCAPS
The **lTransformCaps** member is valid.

D3DDD_TRICAPS

The **dpcTriCaps** member is valid.

lFVFCaps
Combination of constants of the **CONST_D3DFVFCAPSFLAGS** that describe the vertex formats supported by this device.

lMaxAnisotropy
Maximum valid value for the D3DRENDERSTATE_ANISOTROPY render state.

lMaxBufferSize
Maximum size of the execute buffer for this device. If this member is 0, the application can use any size.

lMinStippleHeight, lMaxStippleHeight
Minimum and maximum height of the stipple pattern for this device.

lMinStippleWidth, lMaxStippleWidth
Minimum and maximum width of the stipple pattern for this device.

lMaxTextureAspectRatio
Maximum texture aspect ratio supported by the hardware; this will typically be a power of 2.

lMaxTextureWidth, lMaxTextureHeight
Maximum texture width and height for this device.

lMaxTextureRepeat
Full range of the integer (non-fractional) bits of the post-normalized texture indices. If the D3DDEVCAPS_TEXREPEATNOTSCALEDBYSIZE bit is set, the device defers scaling by the texture size until after the texture address mode is applied. If it isn't set, the device scales the texture indices by the texture size (largest level-of-detail) prior to interpolation.

lMaxVertexCount
Maximum vertex count for this device.

lMinTextureWidth, lMinTextureHeight
Minimum texture width and height for this device.

lStencilCaps
Constants of the **CONST_D3DSTENCILCAPSFLAGS** enumeration specifying supported stencil-buffer operations. Stencil operations are assumed to be valid for all three stencil-buffer operation render states (D3DRENDERSTATE_STENCILFAIL, D3DRENDERSTATE_STENCILPASS, and D3DRENDERSTATE_STENCILFAILZFAIL).

D3DSTENCILCAPS_DECR
The **D3DSTENCILOP_DECR** operation is supported.

D3DSTENCILCAPS_DECRSAT
The **D3DSTENCILOP_DECRSAT** operation is supported.

D3DSTENCILCAPS_INCR
The **D3DSTENCILOP_INCR** operation is supported.

D3DSTENCILCAPS_INCRSAT
The **D3DSTENCILOP_INCRSAT** operation is supported.

D3DSTENCILCAPS_INVERT
The **D3DSTENCILOP_INVERT** operation is supported.

D3DSTENCILCAPS_KEEP
The **D3DSTENCILOP_KEEP** operation is supported.

D3DSTENCILCAPS_REPLACE
The **D3DSTENCILOP_REPLACE** operation is supported.

D3DSTENCILCAPS_ZERO
The **D3DSTENCILOP_ZERO** operation is supported.

lTextureOpCaps
Combination of constants of the **CONST_D3DTEXOPCAPSFLAGS** enumeration describing the texture operations supported by this device. The following flags are defined:

D3DTEXOPCAPS_ADD
The **D3DTOP_ADD** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSIGNED
The **D3DTOP_ADDSIGNED** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSIGNED2X
The **D3DTOP_ADDSIGNED2X** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSMOOTH
The **D3DTOP_ADDSMOOTH** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDCURRENTALPHA
The **D3DTOP_BLENDCURRENTALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDDIFFUSEALPHA
The **D3DTOP_BLENDDIFFUSEALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDFACTORALPHA
The **D3DTOP_BLENDFACTORALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDTEXTUREALPHA
The **D3DTOP_BLENDTEXTUREALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDTEXTUREALPHAPM
The **D3DTOP_BLENDTEXTUREALPHAPM** texture blending operation is supported by this device.

D3DTEXOPCAPS_BUMPENVMAP
The **D3DTOP_BUMPENVMAP** texture blending operation is supported by this device.

D3DTEXOPCAPS_BUMPENVMAPLUMINANCE
The **D3DTOP_BUMPENVMAPLUMINANCE** texture blending operation is supported by this device.

D3DTEXOPCAPS_DISABLE
The **D3DTOP_DISABLE** texture blending operation is supported by this device.

D3DTEXOPCAPS_DOTPRODUCT3
The **D3DTOP_DOTPRODUCT3** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE
The **D3DTOP_MODULATE** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE2X
The **D3DTOP_MODULATE2X** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE4X
The **D3DTOP_MODULATE4X** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR
The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA
The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR
The **D3DTOP_MODULATEINVALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA
The **D3DTOP_MODULATEINVCOLOR_ADDALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_PREMODULATE
The **D3DTOP_PREMODULATE** texture blending operation is supported by this device.

D3DTEXOPCAPS_SELECTARG1
The **D3DTOP_SELECTARG1** texture blending operation is supported by this device.

D3DTEXOPCAPS_SELECTARG2
The **D3DTOP_SELECTARG2** texture blending operation is supported by this device.

D3DTEXOPCAPS_SUBTRACT
The **D3DTOP_SUBTRACT** texture blending operation is supported by this device.

lTransformCaps
One of the constants of the **CONST_D3DTRANSFORMCAPS** enumeration, specifying the transformation capabilities of the device.

nMaxSimultaneousTextures

Maximum number of textures that can be simultaneously bound to the texture blending stages for this device. See remarks.

nMaxTextureBlendStages
Maximum number of texture blending stages supported by this device.

Remarks

The **wMaxTextureBlendStages** and **wMaxSimultaneousTextures** members might seem very similar at first glance, but they contain different information. The **wMaxTextureBlendStages** member contains the total number of texture-blending stages supported by the current device, and the **wMaxSimultaneousTextures** member describes how many of those stages can have textures bound to them by using the **Direct3DDevice3.SetTexture** method.

See Also

CONST_D3DCOLORMODEL, **D3DFINDDEVICERESULT2**, **D3DLIGHTINGCAPS**, **D3DPRIMCAPS**, **CONST_D3DTRANSFORMCAPS**

D3DFINDDEVICERESULT2

[#] [This is preliminary documentation and subject to change.]

The **D3DFINDDEVICERESULT2** type identifies a device an application has found by calling the **Direct3D3.FindDevice** method.

```
Type D3DFINDDEVICERESULT
   ddHwDesc As D3DDEVICEDESC
   ddSwDesc As D3DDEVICEDESC
   strGuid As String
End Type
```

ddHwDesc and **ddSwDesc**
D3DDEVICEDESC types describing the hardware and software devices that were found.

strGuid
Globally unique identifier (GUID) of the device that was found.

See Also

D3DFINDDEVICESEARCH

[#] IDH__dx_D3DFINDDEVICERESULT2_d3d_vb

D3DFINDDEVICESEARCH

[#] [This is preliminary documentation and subject to change.]

The **D3DFINDDEVICESEARCH** type specifies the characteristics of a device an application wants to find. This type is used in calls to the **Direct3D3.FindDevice** method.

```
Type D3DFINDDEVICESEARCH
   dcmColorModel As CONST_D3DCOLORMODEL
   dpcPrimCaps As D3DPRIMCAPS
   lCaps As Long
   lFlags As CONST_D3DFINDDEVICESEARCHFLAGS
   lHardware As CONST_DBOOLFLAGS
   strGuid As String
End Type
```

dcmColorModel
One of the constants of the **CONST_D3DCOLORMODEL** enumeration, specifying whether the device to find should use the ramp or RGB color model.

dpcPrimCaps
Specifies a **D3DPRIMCAPS** type defining the device's capabilities for each primitive type.

lCaps
Reserved.

lFlags
Flags defining the type of device the application wants to find. This member can be one or more of the following constants of the **CONST_D3DFINDDEVICESEARCHFLAGS** enumeration:

D3DFDS_ALPHACMPCAPS
Match the **lAlphaCmpCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

D3DFDS_COLORMODEL
Match the color model specified in the **dcmColorModel** member of this structure.

D3DFDS_DSTBLENDCAPS
Match the **lDestBlendCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

D3DFDS_GUID
Match the globally unique identifier (GUID) specified in the **guid** member of this structure.

D3DFDS_HARDWARE
Match the hardware or software search specification given in the **bHardware** member of this structure.

[#] IDH__dx_D3DFINDDEVICESEARCH_d3d_vb

D3DFDS_LINES
Match the **D3DPRIMCAPS** type specified by the **dpcLineCaps** member of the **D3DDEVICEDESC** structure.

D3DFDS_MISCCAPS
Match the **lMiscCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

D3DFDS_RASTERCAPS
Match the **lRasterCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

D3DFDS_SHADECAPS
Match the **lShadeCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

D3DFDS_SRCBLENDCAPS
Match the **lSrcBlendCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TEXTUREBLENDCAPS
Match the **lTextureBlendCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TEXTURECAPS
Match the **lTextureCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TEXTUREFILTERCAPS
Match the **lTextureFilterCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

D3DFDS_TRIANGLES
Match the **D3DPRIMCAPS** type specified by the **dpcTriCaps** member of the **D3DDEVICEDESC** structure.

D3DFDS_ZCMPCAPS
Match the **lZCmpCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this structure.

lHardware
Constant of the **CONST_DBOOLFLAGS** enumeration specifying whether the device to find is implemented as hardware or software. If this member is True, the device to search for has hardware rasterization and may also provide other hardware acceleration. Applications that use this flag should set the D3DFDS_HARDWARE bit in the **lFlags** member.

strGuid
Globally unique identifier (GUID) of the device to find.

See Also

D3DFINDDEVICERESULT2

D3DHVERTEX

[This is preliminary documentation and subject to change.]

The **D3DHVERTEX** type defines a homogeneous vertex used when the application is supplying screen coordinate data that needs clipping. This type is part of the **D3DTRANSFORMDATA** structure.

```
Type D3DHVERTEX
    hx As Single
    hy As Single
    hz As Single
    lFlags As Long
End Type
```

hx, **hy**, and **hz**
Values describing transformed homogeneous coordinates. These coordinates define the vertex.

lFlags
Flags defining the clip status of the homogeneous vertex. This member can be one or more of the flags described in the **lClip** member of the **D3DTRANSFORMDATA** type.

See Also

D3DTLVERTEX, **D3DVERTEX**

D3DLIGHT2

[This is preliminary documentation and subject to change.]

The **D3DLIGHT2** type defines the light type in calls to methods such as **Direct3DLight.SetLight** and **Direct3DLight.GetLight**.

```
Type D3DLIGHT2
    attenuation0 As Single
    attenuation1 As Single
    attenuation2 As Single
    color As D3DCOLORVALUE
    direction As D3DVECTOR
    dltType As CONST_D3DLIGHTTYPE
    falloff As Single
    lFlags As CONST_D3DLIGHT2FLAGS
    phi As Single
    position As D3DVECTOR
```

IDH__dx_D3DHVERTEX_d3d_vb
IDH__dx_D3DLIGHT2_d3d_vb

```
    range As Single
    theta As Single
End Type
```

attenuation0, **attenuation1**, and **attenuation2**
Values specifying how a light's intensity changes over distance. (Attenuation does not affect directional lights.) In the **D3DLIGHT2** type these values are interpreted differently than they were for the **D3DLIGHT** structure.

color
Color of the light. This member is a **D3DCOLORVALUE** structure. In ramp mode, the color is converted to a gray scale.

direction
Direction the light is pointing in world space. This member only has meaning for directional and spotlights. This vector need not be normalized but it should have a nonzero length.

dltType
Type of the light source. This value is one of the following members of the **CONST_D3DLIGHTTYPE** enumeration:

D3DLIGHT_DIRECTIONAL
Light is a directional source. This is equivalent to using a point light source at an infinite distance.

D3DLIGHT_PARALLELPOINT
Light is a parallel point source. This light type acts like a directional light except its direction is the vector going from the light position to the origin of the geometry it is illuminating.

D3DLIGHT_POINT
Light is a point source. The light has a position in space and radiates light in all directions.

D3DLIGHT_SPOT
Light is a spotlight source. This light is something like a point light except that the illumination is limited to a cone. This light type has a direction and several other parameters which determine the shape of the cone it produces. For information about these parameters, see the **D3DLIGHT2** type.

falloff
Decrease in illumination between a spotlight's inner cone (the angle specified by the **theta** member) and the outer edge of the outer cone (the angle specified by the **phi** member).

The effect of falloff on the lighting is subtle. Furthermore, a small performance penalty is incurred by shaping the falloff curve. For these reasons, most developers set this value to 1.0.

lFlags
A combination of the following performance-related constants of the **CONST_D3DLIGHT2FLAGS** enumeration.

D3DLIGHT_ACTIVE

Enables the light. This flag must be set to enable the light; if it is not set, the light is ignored.

D3DLIGHT_NO_SPECULAR
Turns off specular highlights for the light.

phi
Angle, in radians, defining the outer edge of the spotlight's outer cone. Points outside this cone are not lit by the spotlight. This value must be between 0 and pi.

position
Position of the light in world space. This member has no meaning for directional lights and is ignored in that case.

range
Distance beyond which the light has no effect. The maximum allowable value for this member is D3DLIGHT_RANGE_MAX, which is defined as the square root of FLT_MAX. This member does not affect directional lights.

theta
Angle, in radians, of the spotlight's inner cone — that is, the fully illuminated spotlight cone. This value must be between 0 and the value specified by the **phi** member.

Remarks

For more information about lights, see **Direct3DLight**.

See Also

CONST_D3DLIGHTSTATETYPE

D3DLIGHTDATA

[#] [This is preliminary documentation and subject to change.]

The **D3DLIGHTDATA** type describes the points to be lit and resulting colors in calls to the **Direct3DViewport3.LightElements** method.

```
Type D3DLIGHTDATA
   lInSize As Long
   lOutSize As Long
End Type
```

lInSize
Amount to skip from one input element to the next. This allows the application to store extra data inline with the element.

lOutSize

[#] IDH__dx_D3DLIGHTDATA_d3d_vb

Amount to skip from one output color to the next. This allows the application to store extra data inline with the color.

D3DLIGHTINGCAPS

[This is preliminary documentation and subject to change.]

The **D3DLIGHTINGCAPS** type describes the lighting capabilities of a device. This type is a member of the **D3DDEVICEDESC** structure.

```
Type D3DLIGHTINGCAPS
   lCaps As CONST_D3DLIGHTCAPSFLAGS
   lLightingModel As CONST_D3DLIGHTINGMODELFLAGS
   lNumLights As Long
End Type
```

lCaps

Flags describing the capabilities of the lighting module. The following constants of the **CONST_D3DLIGHTCAPSFLAGS** enumeration are defined:

D3DLIGHTCAPS_DIRECTIONAL
Supports directional lights.

D3DLIGHTCAPS_PARALLELPOINT
Supports parallel point lights.

D3DLIGHTCAPS_POINT
Supports point lights.

D3DLIGHTCAPS_SPOT
Supports spotlights.

lLightingModel

Flags defining whether the lighting model is RGB or monochrome. The following constants of the **CONST_D3DLIGHTINGMODELFLAGS** enumeration are defined:

D3DLIGHTINGMODEL_MONO
Monochromatic lighting model.

D3DLIGHTINGMODEL_RGB
RGB lighting model.

lNumLights

Number of lights that can be handled.

D3DLVERTEX

[This is preliminary documentation and subject to change.]

IDH__dx_D3DLIGHTINGCAPS_d3d_vb
IDH__dx_D3DLVERTEX_d3d_vb

The **D3DLVERTEX** type defines an untransformed and lit vertex (model coordinates with color). An application should use this type when the vertex transformations will be handled by Direct3D. This type contains only data and a color that would be filled by software lighting.

```
Type D3DLVERTEX
    color As Long
    specular As Long
    tu As Single
    tv As Single
    x As Single
    y As Single
    z As Single
End Type
```

color and **specular**
Values specifying the color and specular component of the vertex.

tu and **tv**
Values specifying the texture coordinates of the vertex.

x, **y**, and **z**
Values specifying the model coordinates of the vertex.

See Also

D3DTLVERTEX, **D3DVERTEX**

D3DMATERIAL

[#] [This is preliminary documentation and subject to change.]

The **D3DMATERIAL** type specifies material properties in calls to the **Direct3DMaterial3.GetMaterial** and **Direct3DMaterial3.SetMaterial** methods.

```
Type D3DMATERIAL
    ambient As D3DCOLORVALUE
    diffuse As D3DCOLORVALUE
    emissive As D3DCOLORVALUE
    hTexture As Long
    lRampSize As Long
    power As Single
    specular As D3DCOLORVALUE
End Type
```

diffuse, **ambient**, **specular**, and **emissive**

[#] IDH__dx_D3DMATERIAL_d3d_vb

Values specifying the diffuse color, ambient color, specular color, and emissive color of the material, respectively. These values are **D3DCOLORVALUE** types.

hTexture
Handle to the texture map for use by a ramp device. This member can be zero to indicate that the material does not use a texture or when the material is being used with a device other than the ramp device.

lRampSize
Size of the color ramp. For the monochromatic (ramp) driver, this value should be 1 for materials assigned to the background.

power
Value specifying the sharpness of specular highlights.

Remarks

The texture handle specified by the **hTexture** member is acquired from Direct3D by loading a texture into the device. The texture handle may be used only when it has been loaded into the device. This texture handle is only used by the legacy ramp device, which is not supported by interfaces introduced in DirectX 6.0, such as the new **Direct3DDevice3** interface.

To turn off specular highlights for a material, you must set the **power** member to 0—simply setting the specular color components to 0 is not enough.

See Also

Direct3DMaterial3.GetMaterial, **Direct3DMaterial3.SetMaterial**

D3DMATRIX

[#] [This is preliminary documentation and subject to change.]

The **D3DMATRIX** type describes a matrix.

```
Type D3DMATRIX
    rc11 As Single
    rc12 As Single
    rc13 As Single
    rc14 As Single
    rc21 As Single
    rc22 As Single
    rc23 As Single
    rc24 As Single
    rc31 As Single
    rc32 As Single
    rc33 As Single
```

[#] IDH__dx_D3DMATRIX_d3d_vb

```
    rc34 As Single
    rc41 As Single
    rc42 As Single
    rc43 As Single
    rc44 As Single
End Type
```

Remarks

In Direct3D, the **rc34** element of a projection matrix cannot be a negative number. If your application needs to use a negative value in this location, it should scale the entire projection matrix by -1, instead.

D3DPRIMCAPS

[#] [This is preliminary documentation and subject to change.]

The **D3DPRIMCAPS** type defines the capabilities for each primitive type. This type is used when creating a device and when querying the capabilities of a device. This type defines several members in the **D3DDEVICEDESC** structure.

```
Type D3DPrimCaps
    lAlphaCmpCaps As CONST_D3DCAPSCMP
    lDestBlendCaps As CONST_D3DCAPSBLEND
    lMiscCaps As CONST_D3DCAPSMISC
    lRasterCaps As CONST_D3DCAPSRASTER
    lShadeCaps As CONST_D3DCAPSSHADE
    lSrcBlendCaps As CONST_D3DCAPSBLEND
    lStippleHeight As Long
    lStippleWidth As Long
    lTextureAddressCaps As CONST_D3DCAPSTEXTUREADDRESS
    lTextureBlendCaps As CONST_D3DCAPSTEXTUREBLEND
    lTextureCaps As CONST_D3DCAPSTEXTURE
    lTextureFilterCaps As CONST_D3DCAPSTEXTUREFILTER
    lZCmpCaps As CONST_D3DCAPSCMP
End Type
```

lAlphaCmpCaps

Alpha-test comparison functions that the driver can perform. If this member contains only the D3DPCMPCAPS_ALWAYS capability or only the D3DPCMPCAPS_NEVER capability, the driver does not support alpha tests. Otherwise, the flags identify the individual comparisons that are supported for alpha testing. This member can be one or more of the following constants of the **CONST_D3DCAPSCMP** enumeration:

D3DPCMPCAPS_ALWAYS

[#] IDH__dx_D3DPRIMCAPS_d3d_vb

Always pass the z test.

D3DPCMPCAPS_EQUAL
Pass the z test if the new z equals the current z.

D3DPCMPCAPS_GREATER
Pass the z test if the new z is greater than the current z.

D3DPCMPCAPS_GREATEREQUAL
Pass the z test if the new z is greater than or equal to the current z.

D3DPCMPCAPS_LESS
Pass the z test if the new z is less than the current z.

D3DPCMPCAPS_LESSEQUAL
Pass the z test if the new z is less than or equal to the current z.

D3DPCMPCAPS_NEVER
Always fail the z test.

D3DPCMPCAPS_NOTEQUAL
Pass the z test if the new z does not equal the current z.

lDestBlendCaps
Constants of the **CONST_D3DCAPSBLEND** enumeration describing the destination blending capabilities. This member can be one or more of the following constants of the **CONST_D3DCAPSBLEND** enumeration. (The RGBA values of the source and destination are indicated with the subscripts *s* and *d*.)

D3DPBLENDCAPS_BOTHINVSRCALPHA
Source blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$) and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden.

D3DPBLENDCAPS_BOTHSRCALPHA
Source blend factor is (A_s, A_s, A_s, A_s) and destination blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$); the destination blend selection is overridden.

D3DPBLENDCAPS_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DPBLENDCAPS_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DPBLENDCAPS_INVDESTALPHA
Blend factor is ($1-A_d$, $1-A_d$, $1-A_d$, $1-A_d$).

D3DPBLENDCAPS_INVDESTCOLOR
Blend factor is ($1-R_d$, $1-G_d$, $1-B_d$, $1-A_d$).

D3DPBLENDCAPS_INVSRCALPHA
Blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$).

D3DPBLENDCAPS_INVSRCCOLOR
Blend factor is ($1-R_d$, $1-G_d$, $1-B_d$, $1-A_d$).

D3DPBLENDCAPS_ONE
Blend factor is (1, 1, 1, 1).

D3DPBLENDCAPS_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DPBLENDCAPS_SRCALPHASAT
Blend factor is (f, f, f, 1); f = min(A_s, 1-A_d).

D3DPBLENDCAPS_SRCCOLOR
Blend factor is (R_s, G_s, B_s, A_s).

D3DPBLENDCAPS_ZERO
Blend factor is (0, 0, 0, 0).

lMiscCaps
General capabilities for this primitive. This member can be one or more of the following constants of the **CONST_D3DCAPSMISC** enumeration:

D3DPMISCCAPS_CONFORMANT
The device conforms to the OpenGL standard.

D3DPMISCCAPS_CULLCCW
The driver supports counterclockwise culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CCW constant of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_CULLCW
The driver supports clockwise triangle culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CW constant of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_CULLNONE
The driver does not perform triangle culling. This corresponds to the D3DCULL_NONE constant of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_LINEPATTERNREP
Not supported.

D3DPMISCCAPS_MASKPLANES
The device can perform a bitmask of color planes.

D3DPMISCCAPS_MASKZ
The device can enable and disable modification of the depth-buffer on pixel operations.

lRasterCaps
Information on raster-drawing capabilities. This member can be one or more of the following constants of the **CONST_D3DCAPSRASTER** ENUMERATION:

D3DPRASTERCAPS_ANISOTROPY
The device supports anisotropic filtering. For more information, see D3DRENDERSTATE_ANISOTROPY in the **CONST_D3DRENDERSTATETYPE** structure.

D3DPRASTERCAPS_ANTIALIASEDGES
The device can antialias lines forming the convex outline of objects. For more information, see D3DRENDERSTATE_EDGEANTIALIAS in the **CONST_D3DRENDERSTATETYPE** enumeration.

D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT

The device supports antialiasing that is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur. For more information, see the **CONST_D3DANTIALIASMODE** enumeration.

D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT
The device supports antialiasing that is not dependent on the sort order of the polygons. For more information, see the **CONST_D3DANTIALIASMODE** enumeration.

D3DPRASTERCAPS_DITHER
The device can dither to improve color resolution.

D3DPRASTERCAPS_FOGRANGE
The device supports range-based fog. In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene.

D3DPRASTERCAPS_FOGTABLE
The device calculates the fog value by referring to a lookup table containing fog values that are indexed to the depth of a given pixel.

D3DPRASTERCAPS_FOGVERTEX
The device calculates the fog value during the lighting operation, places the value into the alpha component given for the **specular** member of the **D3DTLVERTEX** structure, and interpolates the fog value during rasterization.

D3DPRASTERCAPS_MIPMAPLODBIAS
The device supports level-of-detail (LOD) bias adjustments. These bias adjustments enable an application to make a mipmap appear crisper or less sharp than it normally would. For more information about LOD bias in mipmaps, see D3DRENDERSTATE_MIPMAPLODBIAS**.**

D3DPRASTERCAPS_PAT
The driver can perform patterned drawing (lines or fills with the D3DRENDERSTATE_STIPPLEPATTERN render state) for the primitive being queried.

D3DPRASTERCAPS_ROP2
The device can support raster operations other than R2_COPYPEN.

D3DPRASTERCAPS_STIPPLE
The device can stipple polygons to simulate translucency.

D3DPRASTERCAPS_SUBPIXEL
The device performs subpixel placement of z, color, and texture data, rather than working with the nearest integer pixel coordinate. This helps avoid bleed-through due to z imprecision, and jitter of color and texture values for pixels. Note that there is no corresponding state that can be enabled and disabled; the device either performs subpixel placement or it does not, and this bit is present only so that the Direct3D client will be better able to determine what the rendering quality will be.

D3DPRASTERCAPS_SUBPIXELX

The device is subpixel accurate along the x-axis only and is clamped to an integer y-axis scan line. For information about subpixel accuracy, see D3DPRASTERCAPS_SUBPIXEL.

D3DPRASTERCAPS_TRANSLUCENTSORTINDEPENDENT
The device supports translucency that is not dependent on the sort order of the polygons. For more information, see the D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT.

D3DPRASTERCAPS_WBUFFER
The device supports depth buffering using w.

D3DPRASTERCAPS_WFOG
The device supports w-based fog. W-based fog is used when a perspective projection matrix is specified, but affine projections will still use z-based fog. The system considers a projection matrix that contains a non-zero value in the [3][4] element to be a perspective projection matrix.

D3DPRASTERCAPS_XOR
The device can support **XOR** operations. If this flag is not set but D3DPRIM_RASTER_ROP2 is set, then **XOR** operations must still be supported.

D3DPRASTERCAPS_ZBIAS
The device supports z-bias values. These are integer values assigned to polygons that allow physically coplanar polygons to appear separate. For more information, see D3DRENDERSTATE_ZBIAS in the **CONST_D3DRENDERSTATETYPE** enumeration.

D3DPRASTERCAPS_ZBUFFERLESSHSR
The device can perform hidden-surface removal (HSR) without requiring the application to sort polygons, and without requiring the allocation of a depth-buffer. This leaves more video memory for textures. The method used to perform hidden-surface removal is hardware-dependent and is transparent to the application.

Z-bufferless HSR is performed if no depth-buffer surface is attached to the rendering-target surface and the depth-buffer comparison test is enabled (that is, when the state value associated with the D3DRENDERSTATE_ZENABLE enumeration constant is set to True).

D3DPRASTERCAPS_ZTEST
The device can perform z-test operations. This effectively renders a primitive and indicates whether any z pixels would have been rendered.

lShadeCaps
Shading operations that the device can perform. It is assumed, in general, that if a device supports a given command (such as D3DOP_TRIANGLE) at all, it supports the D3DSHADE_FLAT mode (as specified in the **CONST_D3DSHADEMODE** enumeration). This flag specifies whether the driver can also support Gouraud and Phong shading and whether alpha color components are supported for each of the three color-generation modes. When alpha components are not supported in a given mode, the alpha value of colors

generated in that mode is implicitly 255. This is the maximum possible alpha (that is, the alpha component is at full intensity).

With the monochromatic shade modes, the blue channel of the specular component is interpreted as a white intensity. (This is controlled by the D3DRENDERSTATE_MONOENABLE render state.)

The color, specular highlights, fog, and alpha interpolants of a triangle each have capability flags that an application can use to find out how they are implemented by the device driver. These are modified by the shade mode, color model, and by whether the alpha component of a color is blended or stippled.

This member can be one or more of the following constants of the **CONST_D3DCAPSSHADE** enumeration:

D3DPSHADECAPS_ALPHAFLATBLEND
D3DPSHADECAPS_ALPHAFLATSTIPPLED
Device can support an alpha component for flat blended and stippled transparency, respectively (the D3DSHADE_FLAT state for the **CONST_D3DSHADEMODE** enumeration). In these modes, the alpha color component for a primitive is provided as part of the color for the first vertex of the primitive.

D3DPSHADECAPS_ALPHAGOURAUDBLEND
D3DPSHADECAPS_ALPHAGOURAUDSTIPPLED
Device can support an alpha component for Gouraud blended and stippled transparency, respectively (the D3DSHADE_GOURAUD state for the **CONST_D3DSHADEMODE** enumeration). In these modes, the alpha color component for a primitive is provided at vertices and interpolated across a face along with the other color components.

D3DPSHADECAPS_ALPHAPHONGBLEND
D3DPSHADECAPS_ALPHAPHONGSTIPPLED
Device can support an alpha component for Phong blended and stippled transparency, respectively (the D3DSHADE_PHONG state for the **CONST_D3DSHADEMODE** enumeration). In these modes, vertex parameters are reevaluated on a per-pixel basis, applying lighting effects for the red, green, and blue color components. Phong shading is not currently supported.

D3DPSHADECAPS_COLORFLATMONO
D3DPSHADECAPS_COLORFLATRGB
Device can support colored flat shading in color models, respectively. In these modes, the color component for a primitive is provided as part of the color for the first vertex of the primitive. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORGOURAUDMONO
D3DPSHADECAPS_COLORGOURAUDRGB
Device can support colored Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, the color component for a primitive is provided at vertices and interpolated across a

face along with the other color components. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORPHONGMONO
D3DPSHADECAPS_COLORPHONGRGB
Device can support colored Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, vertex parameters are reevaluated on a per-pixel basis. Lighting effects are applied for the red, green, and blue color components in RGB mode, and for the blue component only for monochromatic mode. Phong shading is not currently supported.

D3DPSHADECAPS_FOGFLAT
D3DPSHADECAPS_FOGGOURAUD
D3DPSHADECAPS_FOGPHONG
Device can support fog in the flat, Gouraud, and Phong shading models, respectively. Phong shading is not currently supported.

D3DPSHADECAPS_SPECULARFLATMONO
D3DPSHADECAPS_SPECULARFLATRGB
Device can support specular highlights in flat shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARGOURAUDMONO
D3DPSHADECAPS_SPECULARGOURAUDRGB
Device can support specular highlights in Gouraud shading in color models, respectively.

D3DPSHADECAPS_SPECULARPHONGMONO
D3DPSHADECAPS_SPECULARPHONGRGB
Device can support specular highlights in Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. Phong shading is not currently supported.

lSrcBlendCaps
Source blending capabilities. This member can be the same capabilities that are defined for the **lDestBlendCaps** member.

lStippleWidth and **lStippleHeight**
Maximum width and height of the supported stipple (up to 32×32).

lTextureAddressCaps
Texture-addressing capabilities. This member can be one or more of the following constants of the **CONST_D3DCAPSTEXTUREADDRESS** enumeration:

D3DPTADDRESSCAPS_BORDER
Device supports setting coordinates outside the range [0.0, 1.0] to the border color, as specified by the D3DRENDERSTATE_BORDERCOLOR render state. This ability corresponds to the **D3DTADDRESS_BORDER** texture-addressing mode.

D3DPTADDRESSCAPS_CLAMP
Device can clamp textures to addresses.

D3DPTADDRESSCAPS_INDEPENDENTUV
Device can separate the texture-addressing modes of the u and v coordinates of the texture. This ability corresponds to the D3DRENDERSTATE_TEXTUREADDRESSU and D3DRENDERSTATE_TEXTUREADDRESSV render-state values.

D3DPTADDRESSCAPS_MIRROR
Device can mirror textures to addresses.

D3DPTADDRESSCAPS_WRAP
Device can wrap textures to addresses.

lTextureBlendCaps
Texture-blending capabilities. See the **CONST_D3DTEXTUREBLEND** enumeration for discussions of the various texture-blending modes. This member can be one or more of the following constants of the **CONST_D3DCAPSTEXTUREBLEND** enumeration:

D3DPTBLENDCAPS_ADD
Supports the additive texture-blending mode, in which the Gouraud interpolants are added to the texture lookup with saturation semantics. This capability corresponds to the D3DTBLEND_ADD member of the **CONST_D3DTEXTUREBLEND** enumeration.

D3DPTBLENDCAPS_COPY
Copy mode texture-blending (D3DTBLEND_COPY from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_DECAL
Decal texture-blending mode (D3DTBLEND_DECAL from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_DECALALPHA
Decal-alpha texture-blending mode (D3DTBLEND_DECALALPHA from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_DECALMASK
Decal-mask texture-blending mode (D3DTBLEND_DECALMASK from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_MODULATE
Modulate texture-blending mode (D3DTBLEND_MODULATE from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_MODULATEALPHA
Modulate-alpha texture-blending mode (D3DTBLEND_MODULATEALPHA from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_MODULATEMASK
Modulate-mask texture-blending mode (D3DTBLEND_MODULATEMASK from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

lTextureCaps
Miscellaneous texture-mapping capabilities. This member can be one or more of the following constants of the **CONST_D3DCAPSTEXTURE** enumeration:

D3DPTEXTURECAPS_ALPHA
Supports RGBA textures in the D3DTEX_DECAL and D3DTEX_MODULATE texture filtering modes. If this capability is not set, then only RGB textures are supported in those modes. Regardless of the setting of this flag, alpha must always be supported in D3DTEX_DECAL_MASK, D3DTEX_DECAL_ALPHA, and D3DTEX_MODULATE_ALPHA filtering modes whenever those filtering modes are available.

D3DPTEXTURECAPS_ALPHAPALETTE
Supports palettized texture surfaces whose palettes contain alpha information (see DDPCAPS_ALPHA in the **DDCAPS** structure).

D3DPTEXTURECAPS_BORDER
Superseded by D3DPTADDRESSCAPS_BORDER.

D3DPTEXTURECAPS_PERSPECTIVE
Perspective correction is supported.

D3DPTEXTURECAPS_POW2
All nonmipmapped textures must have widths and heights specified as powers of two if this flag is set. (Note that all mipmapped textures must always have dimensions that are powers of two.)

D3DPTEXTURECAPS_SQUAREONLY
All textures must be square.

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE
Texture indices are not scaled by the texture size prior to interpolation.

D3DPTEXTURECAPS_TRANSPARENCY
Texture transparency is supported. (Only those texels that are not the current transparent color are drawn.)

lTextureFilterCaps
Texture-map filtering capabilities. General texture filtering flags reflect which texture filtering modes you can set for the D3DRENDERSTATE_TEXTUREMAG, D3DRENDERSTATE_TEXTUREMIN render states. Per-stage filtering capabilities reflect which filtering modes are supported for texture stages when performing multiple texture blending with the **Direct3DDevice3** interface. This member can any combination of the following general and per-stage texture filtering constants of the **CONST_D3DCAPSTEXTUREFILTER** enumeration:

General texture filtering flags

D3DPTFILTERCAPS_LINEAR
A weighted average of a 2×2 area of texels surrounding the desired pixel is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DPTFILTERCAPS_LINEARMIPLINEAR
Similar to D3DPTFILTERCAPS_MIPLINEAR, but interpolates between the two nearest mipmaps.

D3DPTFILTERCAPS_LINEARMIPNEAREST

The mipmap chosen is the mipmap whose texels most closely match the size of the pixel to be textured. The D3DFILTER_LINEAR method is then used with the texture.

D3DPTFILTERCAPS_MIPLINEAR
Two mipmaps are chosen whose texels most closely match the size of the pixel to be textured. The D3DFILTER_NEAREST method is then used with each texture to produce two values which are then weighted to produce a final texel value.

D3DPTFILTERCAPS_MIPNEAREST
Similar to D3DPTFILTERCAPS_NEAREST, but uses the appropriate mipmap for texel selection.

D3DPTFILTERCAPS_NEAREST
The texel with coordinates nearest to the desired pixel value is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

Per-stage texture filtering flags

D3DPTFILTERCAPS_MAGFAFLATCUBIC
The device supports per-stage flat-cubic filtering for magnifying textures. The flat-cubic magnification filter is represented by the D3DTFG_FLATCUBIC member of the **CONST_D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MAGFANISOTROPIC
The device supports per-stage anisotropic filtering for magnifying textures. The anisotropic magnification filter is represented by the D3DTFG_ANISOTROPIC member of the **D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MAGFGAUSSIANCUBIC
The device supports the per-stage Gaussian-cubic filtering for magnifying textures. The Gaussian-cubic magnification filter is represented by the D3DTFG_GAUSSIANCUBIC member of the **D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MAGFLINEAR
The device supports per-stage bilinear-interpolation filtering for magnifying textures. The bilinear-interpolation magnification filter is represented by the D3DTFG_LINEAR member of the **D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MAGFPOINT
The device supports per-stage point-sampled filtering for magnifying textures. The point-sample magnification filter is represented by the D3DTFG_POINT member of the **D3DTEXTUREMAGFILTER** enumeration.

D3DPTFILTERCAPS_MINFANISOTROPIC
The device supports per-stage anisotropic filtering for minifying textures. The anisotropic minification filter is represented by the D3DTFN_ANISOTROPIC member of the **CONST_D3DTEXTUREMINFILTER** enumeration.

D3DPTFILTERCAPS_MINFLINEAR

The device supports per-stage bilinear-interpolation filtering for minifying textures. The bilinear minification filter is represented by the D3DTFN_LINEAR member of the **D3DTEXTUREMINFILTER** enumeration.

D3DPTFILTERCAPS_MINFPOINT
The device supports per-stage point-sampled filtering for minifying textures. The point-sample minification filter is represented by the D3DTFN_POINT member of the **D3DTEXTUREMINFILTER** enumeration.

D3DPTFILTERCAPS_MIPFLINEAR
The device supports per-stage trilinear-interpolation filtering for mipmaps. The trilinear-interpolation mipmapping filter is represented by the D3DTFP_LINEAR member of the **CONST_D3DTEXTUREMIPFILTER** enumeration.

D3DPTFILTERCAPS_MIPFPOINT
The device supports per-stage point-sampled filtering for mipmaps. The point-sample mipmapping filter is represented by the D3DTFP_POINT member of the **D3DTEXTUREMIPFILTER** enumeration.

lZCmpCaps
Z-buffer comparison functions that the driver can perform. This member can include same constants of the **CONST_D3DCAPSCMP** as defined for the **lAlphaCmpCaps** member.

D3DRECT

[#] [This is preliminary documentation and subject to change.]

The **D3DRECT** type is a rectangle definition.

```
Type D3DRECT {
   x1 As Long
   x2 As Long
   y1 As Long
   y2 As Long
End Type
```

x1 and **y1**
Coordinates of the upper-left corner of the rectangle.

x2 and **y2**
Coordinates of the lower-right corner of the rectangle.

See Also

Direct3DViewport3.Clear

[#] IDH__dx_D3DRECT_d3d_vb

D3DSTATS

[This is preliminary documentation and subject to change.]

The **D3DSTATS** type contains statistics used by the **Direct3DDevice3.GetStats** method.

```
Type D3DSTATS
    lLinesDrawn As Long
    lPointsDrawn As Long
    lSpansDrawn As Long
    lTrianglesDrawn As Long
    lVerticesProcessed As Long
End Type
```

lTrianglesDrawn, **lLinesDrawn**, **lPointsDrawn**, and **lSpansDrawn**
Number of triangles, lines, points, and spans drawn since the device was created.

lVerticesProcessed
Number of vertices processed since the device was created.

See Also

Direct3DDevice3.GetStats

D3DTLVERTEX

[This is preliminary documentation and subject to change.]

The **D3DTLVERTEX** type defines a transformed and lit vertex (screen coordinates with color) for the **D3DLIGHTDATA** structure.

```
Type D3DTLVERTEX
    color As Long
    rhw As Single
    specular As Long
    sx As Single
    sy As Single
    sz As Single
    tu As Single
    tv As Single
End Type
```

color and **specular**
Values describing the color and specular component of the vertex.

IDH__dx_D3DSTATS_d3d_vb
IDH__dx_D3DTLVERTEX_d3d_vb

rhw
Value that is the reciprocal of homogeneous w from homogeneous coordinate (x,y,z,w). This value is often 1 divided by the distance from the origin to the object along the z-axis.

sx, **sy**, and **sz**
Values describing a vertex in screen coordinates. The largest allowable value for **sz** is 1.0, if you want the vertex to be within the range of z-values that are displayed.

tu and **tv**
Values describing the texture coordinates of the vertex.

Remarks

Direct3D uses the current viewport parameters (the **lX**, **lY**, **lWidth**, and **lHeight** members of the **D3DVIEWPORT2** structure) to clip **D3DTLVERTEX** vertices. The system always clips z-coordinates to [0, 1]. To prevent the system from clipping these vertices, use the D3DDP_DONOTCLIP flag in your call to **Direct3DDevice3.Begin**.

See Also

D3DLIGHTDATA, **D3DLVERTEX**, **D3DVERTEX**

D3DTRANSFORMDATA

[#] [This is preliminary documentation and subject to change.]

The **D3DTRANSFORMDATA** type contains information about transformations for the **Direct3DViewport3.TransformVertices** method.

```
Type D3DTRANSFORMDATA
    lClip As CONST_D3DCLIPFLAGS
    lClipIntersection As CONST_D3DCLIPFLAGS
    lClipUnion As CONST_D3DCLIPFLAGS
    rExtent As D3DRECT
End Type
```

lClip
Flags specifying how the vertices are clipped. This member can be one or more of the following values constants of the **CONST_D3DCLIPFLAGS** enumeration:

D3DCLIP_BACK
Clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM
Clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT

[#] IDH__dx_D3DTRANSFORMDATA_d3d_vb

Clipped by the front plane of the viewing frustum.

D3DCLIP_GEN0 through D3DCLIP_GEN5
Application-defined clipping planes.

D3DCLIP_LEFT
Clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT
Clipped by the right plane of the viewing frustum.

D3DCLIP_TOP
Clipped by the top plane of the viewing frustum_dx_viewing_frustum_glos.

lClipIntersection
Flags denoting the intersection of the clip flags. This member can be one or more of the following values constants of the **CONST_D3DCLIPFLAGS** enumeration:

D3DSTATUS_CLIPINTERSECTIONBACK
Logical **AND** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM
Logical **AND** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT
Logical **AND** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through
D3DSTATUS_CLIPINTERSECTIONGEN5
Logical **AND** of the clip flags for application-defined clipping planes.

D3DSTATUS_CLIPINTERSECTIONLEFT
Logical **AND** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT
Logical **AND** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP
Logical **AND** of the clip flags for the vertices compared to the top of the viewing frustum.

lClipUnion
Flags denoting the union of the clip flags. This member can be one or more of the following values constants of the **CONST_D3DCLIPFLAGS** enumeration:

D3DSTATUS_CLIPUNIONBACK
Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM
Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT
Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5
Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT
Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT
Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP
Equal to D3DCLIP_TOP.

rExtent
A **D3DRECT** type that defines the extents of the transformed vertices. Initialize this type to initial extents that the **Direct3DViewport3.TransformVertices** method will adjust if the transformed vertices do not fit. For geometries that are clipped, extents will only include vertices that are inside the viewing volume.

Remarks

All values generated by the transformation module are stored as 16-bit precision values. The clip is treated as an integer bitfield that is set to the inclusive **OR** of the viewing volume planes that clip a given transformed vertex.

See Also

Direct3DViewport3.TransformVertices

D3DVECTOR

[#] [This is preliminary documentation and subject to change.]

The **D3DVECTOR** type defines a vector for many Direct3D and Direct3DRM methods and types.

```
Type D3DVECTOR
    x As Single
    y As Single
    z As Single
End Type
```

x, **y**, and **z**
Values describing the vector.

See Also

D3DLIGHT2

[#] IDH__dx_D3DVECTOR_d3d_vb

D3DVERTEX

[This is preliminary documentation and subject to change.]

The **D3DVERTEX** type defines an untransformed and unlit vertex (model coordinates with normal direction vector).

```
Type D3DVERTEX
   nx As Single
   ny As Single
   nz As Single
   tu As Single
   tv As Single
   x As Single
   y As Single
   z As Single
End Type
```

x, **y**, and **z**
Values describing the homogeneous coordinates of the vertex.

nx, **ny**, and **nz**
Values describing the normal coordinates of the vertex.

tu and **tv**
Values describing the texture coordinates of the vertex.

See Also

D3DLVERTEX, **D3DTLVERTEX**

D3DVERTEXBUFFERDESC

[This is preliminary documentation and subject to change.]

The **D3DVERTEXBUFFERDESC** type describes the properties of a vertex buffer object. This type is used with the **Direct3D3.CreateVertexBuffer** and **Direct3DVertexBuffer.GetVertexBufferDesc** methods.

```
Type D3DVERTEXBUFFERDESC
   lCaps As CONST_D3DVBCAPSFLAGS
   lFVF As CONST_D3DFVFFLAGS
   lNumVertices As Long
End Type
```

lCaps

IDH__dx_D3DVERTEX_d3d_vb
IDH__dx_D3DVERTEXBUFFERDESC_d3d_vb

Capability flags that describe the vertex buffer and identify if the vertex buffer can contain optimized vertex data. This parameter can be any combination of the following constants of the **CONST_D3DVBCAPSFLAGS** enumeration:

(none)
The vertex buffer should be created in whatever memory the driver chooses to allow efficient read operations.

D3DVBCAPS_OPTIMIZED
The vertex buffer contains optimized vertex data. (This flag is not used when creating a new vertex buffer.)

D3DVBCAPS_SYSTEMMEMORY
The vertex buffer should be created in system memory. Use this capability for vertex buffers that will be rendered by using software devices (MMX and RGB devices).

D3DVBCAPS_WRITEONLY
Hints to the system that the application will only write to the vertex buffer. Using this flag enables the driver to choose the best memory location for efficient write operations and rendering. Attempts to read from a vertex buffer that is created with this capability can result in degraded performance.

lFVF
A combination of constants of the **CONST_D3DFVFFLAGS** enumeration that describes the vertex format of the vertices in this buffer.

lNumVertices
The maximum number of vertices that this vertex buffer can contain.

Remarks

Software devices — MMX and RGB devices — cannot render from a video memory (local or non-local) vertex buffer. To render a vertex buffer using a software device, the vertex buffer must exist in system memory. Hardware devices can render from system memory or video memory vertex buffers.

See Also

Vertex Buffer Descriptions, Vertex Buffers

D3DVIEWPORT

[#] [This is preliminary documentation and subject to change.]

The **D3DVIEWPORT** type defines the visible 3-D volume and the 2-D screen area that a 3-D volume projects onto for the **Direct3DViewport3.GetViewport** and **Direct3DViewport3.SetViewport** methods.

For the **Direct3D2** and **Direct3DDevice2** interfaces, this type has been superseded by the **D3DVIEWPORT2** structure.

[#] IDH__dx_D3DVIEWPORT_d3d_vb

```
Type D3DVIEWPORT
   lHeight As Long
   lWidth As Long
   lX As Long
   lY As Long
   maxx As Single
   maxy As Single
   maxz As Single
   minz As Single
   scaleX As Single
   scaleY As Single
End Type
```

lWidth and **lHeight**
Dimensions of the viewport.

lX and **lY**
Coordinates of the top-left corner of the viewport.

maxx, **maxy**, **minz**, and **maxz**
Values describing the maximum and minimum nonhomogeneous coordinates of x, y, and z. Again, the relevant coordinates are the nonhomogeneous coordinates that result from the perspective division.

scaleX and **scaleY**
Values describing how coordinates are scaled. The relevant coordinates here are the nonhomogeneous coordinates that result from the perspective division that projects the vertices onto the w=1 plane.

Remarks

When the viewport is changed, the driver builds a new transformation matrix.

The coordinates and dimensions of the viewport are given relative to the top left of the device.

See Also

Direct3DViewport3.GetViewport, **Direct3DViewport3.SetViewport**

D3DVIEWPORT2

[#] [This is preliminary documentation and subject to change.]

The **D3DVIEWPORT2** type defines the visible 3-D volume and the window dimensions that a 3-D volume projects onto. This type is used by the methods of the **Direct3D2** and **Direct3DDevice2** interfaces, and in particular by the

---

[#] IDH__dx_D3DVIEWPORT2_d3d_vb

Direct3DViewport3.GetViewport2 and **Direct3DViewport3.SetViewport2** methods.

```
Type D3DVIEWPORT2
    clipHeight As Single
    clipWidth As Single
    clipX As Single
    clipY As Single
    lHeight As Long
    lWidth As Long
    lX As Long
    lY As Long
    maxz As Single
    minz As Single
End Type
```

clipWidth and **clipHeight**
Dimensions of the clipping volume projected onto the w=1 plane. Unless you want to render to a subset of the surface, these members can be set to the width and height of the destination surface.

clipX and **clipY**
Coordinates of the top-left corner of the clipping volume.

The relevant coordinates here are the nonhomogeneous coordinates that result from the perspective division that projects the vertices onto the w=1 plane.

lWidth and **lHeight**
Dimensions of the viewport on the render target surface, in pixels. Unless you are rendering only to a subset of the surface, these members should be set to the dimensions of the render target surface.

lX and **lY**
Pixel coordinates of the top-left corner of the viewport on the render target surface. Unless you want to render to a subset of the surface, these members can be set to zero.

minz and **maxz**
Values describing the maximum and minimum nonhomogeneous z-coordinates resulting from the perspective divide and projected onto the w=1 plane. The values in these members must not be identical.

Remarks

The **lX**, **lY**, **lWidth** and **lHeight** members describe the position and dimensions of the viewport on the render target surface. Usually, applications render to the entire target surface; when rendering on a 640x480 surface, these members should be 0, 0, 640, and 480, respectively.

The **clipX**, **clipY**, **clipWidth**, **clipHeight**, **minz**, and **maxz** members define the non-normalized post-perspective 3-D view volume which is visible to the viewer. In most

cases, **clipX** is set to -1.0 and **clipY** is set to the inverse of the viewport's aspect ratio on the target surface, which can be calculated by dividing the **lHeight** member by **lWidth**. Similarly, the **clipWidth** member is typically 2.0 and **clipHeight** is set to twice the aspect ratio set in **lClipY**. The **minz** and **maxz** are usually set to 0.0 and 1.0.

Unlike the **D3DVIEWPORT** structure, **D3DVIEWPORT2** specifies the relationship between the size of the viewport and the window. The coordinates and dimensions of the viewport are given relative to the top left of the device; values increase in the y-direction as you descend the screen.

When the viewport is changed, the driver builds a new transformation matrix.

See Also

Direct3DViewport3.GetViewport2, **Direct3DViewport3.SetViewport2**

DXDRIVERINFO

[#] [This is preliminary documentation and subject to change.]

The **DXDRIVERINFO** type is used in the enumeration methods for DirectDraw, DirectSound and Direct3D to hold driver information.

```
Type DXDRIVERINFO
   strDescription As String
   strGuid As String
   strName As String
End Type
```

strDescription
The textual description of the Direct3D device.

strGuid
The GUID that identifies the Direct3D driver being enumerated.

strName
The name of the Direct3D driver corresponding to this device.

Remarks

This type is also used in DirectDraw and DirectSound.

Enumerations

[This is preliminary documentation and subject to change.]

This section contains information about the following enumerations used with Direct3D Immediate Mode.

[#] IDH__dx_DXDRIVERINFO_d3d_vb

- **CONST_D3D**
- **CONST_D3DANTIALIASMODE**
- **CONST_D3DBLEND**
- **CONST_D3DCAPSBLEND**
- **CONST_D3DCAPSCMP**
- **CONST_D3DCAPSMISC**
- **CONST_D3DCAPSRASTER**
- **CONST_D3DCAPSSHADE**
- **CONST_D3DCAPSTEXTURE**
- **CONST_D3DCAPSTEXTUREADDRESS**
- **CONST_D3DCAPSTEXTUREBLEND**
- **CONST_D3DCAPSTEXTUREFILTER**
- **CONST_D3DCLEARFLAGS**
- **CONST_D3DCLIPFLAGS**
- **CONST_D3DCLIPSTATUSFLAGS**
- **CONST_D3DCMPFUNC**
- **CONST_D3DCOLORMODEL**
- **CONST_D3DCULL**
- **CONST_D3DDEVICEDESCCAPS**
- **CONST_D3DDEVICEDESCFLAGS**
- **CONST_D3DDPFLAGS**
- **CONST_D3DFILLMODE**
- **CONST_D3DFINDDEVICESEARCHFLAGS**
- **CONST_D3DFOGMODE**
- **CONST_D3DFVFCAPSFLAGS**
- **CONST_D3DFVFFLAGS**
- **CONST_D3DIMERR**
- **CONST_D3DLIGHT2FLAGS**
- **CONST_D3DLIGHTCAPSFLAGS**
- **CONST_D3DLIGHTINGMODELFLAGS**
- **CONST_D3DLIGHTSTATETYPE**
- **CONST_D3DLIGHTTYPE**
- **CONST_D3DNEXTFLAGS**
- **CONST_D3DPALFLAGS**
- **CONST_D3DPRIMITIVETYPE**
- **CONST_D3DRENDERSTATETYPE**
- **CONST_D3DSETSTATUSFLAGS**

- **CONST_D3DSHADEMODE**
- **CONST_D3DSTENCILCAPSFLAGS**
- **CONST_D3DSTENCILOP**
- **CONST_D3DTAFLAGS**
- **CONST_D3DTEXOPCAPSFLAGS**
- **CONST_D3DTEXTUREADDRESS**
- **CONST_D3DTEXTUREBLEND**
- **CONST_D3DTEXTUREFILTER**
- **CONST_D3DTEXTUREMAGFILTER**
- **CONST_D3DTEXTUREMINFILTER**
- **CONST_D3DTEXTUREMIPFILTER**
- **CONST_D3DTEXTUREOP**
- **CONST_D3DTEXTURESTAGESTATETYPE**
- **CONST_D3DTRANSFORMCAPS**
- **CONST_D3DTRANSFORMFLAGS**
- **CONST_D3DTRANSFORMSTATETYPE**
- **CONST_D3DVBCAPSFLAGS**
- **CONST_D3DVERTEXTYPE**
- **CONST_D3DVISFLAGS**
- **CONST_D3DVOPFLAGS**
- **CONST_D3DZBUFFERTYPE**

CONST_D3D

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3D** enumeration defines miscellaneous constants.

```
Enum CONST_D3D
   D3DRENDERSTATE_WRAPBIAS = 128
   D3DDP_MAXTEXCOORD =8
   D3DWRAP_U=   1
   D3DWRAP_V=   2
End Enum
```

D3DRENDERSTATE_WRAPBIAS
A convenience value that can be added to a zero-based index for a texture stage to produce a valid D3DRENDERSTATE_WRAP*n* value for use with the **Direct3DDevice3.SetRenderState** and **Direct3DDevice3.SetRenderState** methods.

[#] IDH__dx_CONST_D3D_d3d_vb

D3DDP_MAXTEXCOORD
The maximum number of texture coordinates allowed for a vertex.

D3DWRAP_U and D3DWRAP_V
Enables or disables texture wrapping in the U and V directions. These values are used to set values for the **D3DRENDERSTATE_WRAP0** through **D3DRENDERSTATE_WRAP7** render states.

CONST_D3DANTIALIASMODE

[This is preliminary documentation and subject to change.]

The **CONST_D3DANTIALIASMODE** enumeration defines the supported antialiasing mode for the D3DRENDERSTATE_ANTIALIAS value in the **CONST_D3DRENDERSTATETYPE** enumeration. These values define the settings for antialiasing the edges of primitives.

```
Enum CONST_D3DANTIALIASMODE
   D3DANTIALIAS_NONE         = 0
   D3DANTIALIAS_SORTDEPENDENT = 1
   D3DANTIALIAS_SORTINDEPENDENT = 2
   D3DANTIALIAS_FORCE_DWORD     = &H7FFFFFFF
End Enum
```

D3DANTIALIAS_NONE
No antialiasing is performed. This is the default setting.

D3DANTIALIAS_SORTDEPENDENT
Antialiasing is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur.

D3DANTIALIAS_SORTINDEPENDENT
Antialiasing is not dependent on the sort order of the polygons.

D3DANTIALIAS_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

CONST_D3DBLEND

[This is preliminary documentation and subject to change.]

The **CONST_D3DBLEND** enumeration defines the supported blend mode for the D3DRENDERSTATE_DESTBLEND values in the **CONST_D3DRENDERSTATETYPE** enumeration. In the member descriptions that follow, the RGBA values of the source and destination are indicated with the subscripts *s* and *d*.

IDH__dx_CONST_D3DANTIALIASMODE_d3d_vb
IDH__dx_CONST_D3DBLEND_d3d_vb

```
Enum CONST_D3DBLEND
   D3DBLEND_ZERO          = 1
   D3DBLEND_ONE           = 2
   D3DBLEND_SRCCOLOR      = 3
   D3DBLEND_INVSRCCOLOR   = 4
   D3DBLEND_SRCALPHA      = 5
   D3DBLEND_INVSRCALPHA   = 6
   D3DBLEND_DESTALPHA     = 7
   D3DBLEND_INVDESTALPHA  = 8
   D3DBLEND_DESTCOLOR     = 9
   D3DBLEND_INVDESTCOLOR  = 10
   D3DBLEND_SRCALPHASAT   = 11
   D3DBLEND_BOTHSRCALPHA  = 12
   D3DBLEND_BOTHINVSRCALPHA = 13
   D3DBLEND_FORCE_DWORD   = &H7FFFFFFF
End Enum
```

D3DBLEND_ZERO
Blend factor is (0, 0, 0, 0).

D3DBLEND_ONE
Blend factor is (1, 1, 1, 1).

D3DBLEND_SRCCOLOR
Blend factor is (R_s, G_s, B_s, A_s).

D3DBLEND_INVSRCCOLOR
Blend factor is ($1-R_s$, $1-G_s$, $1-B_s$, $1-A_s$).

D3DBLEND_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DBLEND_INVSRCALPHA
Blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$).

D3DBLEND_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DBLEND_INVDESTALPHA
Blend factor is ($1-A_d$, $1-A_d$, $1-A_d$, $1-A_d$).

D3DBLEND_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DBLEND_INVDESTCOLOR
Blend factor is ($1-R_d$, $1-G_d$, $1-B_d$, $1-A_d$).

D3DBLEND_SRCALPHASAT
Blend factor is (f, f, f, 1); f = min(A_s, $1-A_d$).

D3DBLEND_BOTHSRCALPHA
Not supported.

D3DBLEND_BOTHINVSRCALPHA
Source blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$), and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden.

D3DBLEND_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

Remarks

D3DBLEND_BOTHSRCALPHA is no longer supported in Direct3D versions DX6 and higher. Please explicitly set both D3DBLEND_SRCALPHA and D3DBLEND_INVSRCALPHA separately.

CONST_D3DCAPSBLEND

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DCAPSBLEND** enumeration defines the blending capabilities for a device. These values are combined in the **lSrceBlendCaps** and **lDestBlendCaps** members of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSBLEND
   D3DPBLENDCAPS_BOTHINVSRCALPHA =4096
   D3DPBLENDCAPS_BOTHSRCALPHA =2048
   D3DPBLENDCAPS_DESTALPHA =64
   D3DPBLENDCAPS_DESTCOLOR =256
   D3DPBLENDCAPS_INVDESTALPHA =128
   D3DPBLENDCAPS_INVDESTCOLOR =512
   D3DPBLENDCAPS_INVSRCALPHA =32
   D3DPBLENDCAPS_INVSRCCOLOR =8
   D3DPBLENDCAPS_ONE =2
   D3DPBLENDCAPS_SRCALPHA =16
   D3DPBLENDCAPS_SRCALPHASAT =1024
   D3DPBLENDCAPS_SRCCOLOR =4
   D3DPBLENDCAPS_ZERO =1
End Enum
```

D3DPBLENDCAPS_BOTHINVSRCALPHA
Source blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$) and destination blend factor is (A_s, A_s, A_s, A_s); the destination blend selection is overridden.

D3DPBLENDCAPS_BOTHSRCALPHA
Source blend factor is (A_s, A_s, A_s, A_s) and destination blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$); the destination blend selection is overridden.

D3DPBLENDCAPS_DESTALPHA
Blend factor is (A_d, A_d, A_d, A_d).

D3DPBLENDCAPS_DESTCOLOR
Blend factor is (R_d, G_d, B_d, A_d).

D3DPBLENDCAPS_INVDESTALPHA

[#] IDH__dx_CONST_D3DCAPSBLEND_d3d_vb

Blend factor is ($1-A_d$, $1-A_d$, $1-A_d$, $1-A_d$).

D3DPBLENDCAPS_INVDESTCOLOR
Blend factor is ($1-R_d$, $1-G_d$, $1-B_d$, $1-A_d$).

D3DPBLENDCAPS_INVSRCALPHA
Blend factor is ($1-A_s$, $1-A_s$, $1-A_s$, $1-A_s$).

D3DPBLENDCAPS_INVSRCCOLOR
Blend factor is ($1-R_d$, $1-G_d$, $1-B_d$, $1-A_d$).

D3DPBLENDCAPS_ONE
Blend factor is (1, 1, 1, 1).

D3DPBLENDCAPS_SRCALPHA
Blend factor is (A_s, A_s, A_s, A_s).

D3DPBLENDCAPS_SRCALPHASAT
Blend factor is (f, f, f, 1); f = min(A_s, $1-A_d$).

D3DPBLENDCAPS_SRCCOLOR
Blend factor is (R_s, G_s, B_s, A_s).

D3DPBLENDCAPS_ZERO
Blend factor is (0, 0, 0, 0).

CONST_D3DCAPSCMP

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DCAPSCMP** enumeration defines comparison capabilities for depth-buffer comparisons and alpha-testing. These flags are combined and present in the **lZCmpCaps** and **lAlphaCmpCaps** members of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSCMP
   D3DPCMPCAPS_ALWAYS = 128
   D3DPCMPCAPS_EQUAL = 4
   D3DPCMPCAPS_GREATER = 16
   D3DPCMPCAPS_GREATEREQUAL = 64
   D3DPCMPCAPS_LESS = 2
   D3DPCMPCAPS_LESSEQUAL = 8
   D3DPCMPCAPS_NEVER = 1
   D3DPCMPCAPS_NOTEQUAL = 32
End Enum
```

D3DPCMPCAPS_ALWAYS
Always pass the comparison.

D3DPCMPCAPS_EQUAL
Pass the comparison if the new value equals the current value.

D3DPCMPCAPS_GREATER
Pass the comparison if the new value is greater than the current value.

[#] IDH__dx_CONST_D3DCAPSCMP_d3d_vb

D3DPCMPCAPS_GREATEREQUAL
Pass the comparison if the new value is greater than or equal to the current value.

D3DPCMPCAPS_LESS
Pass the comparison if the new value is less than the current value.

D3DPCMPCAPS_LESSEQUAL
Pass the comparison if the new value is less than or equal to the current value.

D3DPCMPCAPS_NEVER
Always fail the comparison.

D3DPCMPCAPS_NOTEQUAL
Pass the comparison if the new value does not equal the current value.

CONST_D3DCAPSMISC

[This is preliminary documentation and subject to change.]

The **CONST_D3DCAPSMISC** enumeration defines capability flags that are combined and found in the **lMiscCaps** member of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSMISC
   D3DPMISCCAPS_CONFORMANT = 8
   D3DPMISCCAPS_CULLCCW = 64
   D3DPMISCCAPS_CULLCW = 32
   D3DPMISCCAPS_CULLNONE = 16
   D3DPMISCCAPS_LINEPATTERNREP = 4
   D3DPMISCCAPS_MASKPLANES = 1
   D3DPMISCCAPS_MASKZ = 2
End Enum
```

D3DPMISCCAPS_CONFORMANT
The device conforms to the OpenGL standard.

D3DPMISCCAPS_CULLCCW
The driver supports counterclockwise culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CCW constant of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_CULLCW
The driver supports clockwise triangle culling through the D3DRENDERSTATE_CULLMODE state. (This applies only to triangle primitives.) This corresponds to the D3DCULL_CW constant of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_CULLNONE
The driver does not perform triangle culling. This corresponds to the D3DCULL_NONE constant of the **CONST_D3DCULL** enumeration.

D3DPMISCCAPS_LINEPATTERNREP

IDH__dx_CONST_D3DCAPSMISC_d3d_vb

The driver can handle values other than 1 in the **wRepeatFactor** member of the **D3DLINEPATTERN** structure. (This applies only to line-drawing primitives.)

D3DPMISCCAPS_MASKPLANES
The device can perform a bitmask of color planes.

D3DPMISCCAPS_MASKZ
The device can enable and disable modification of the depth-buffer on pixel operations.

CONST_D3DCAPSRASTER

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DCAPSRASTER** enumeration defines rasterization capability flags that are combined and present in the **lRasterCaps** member of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSRASTER
   D3DPRASTERCAPS_ANISOTROPY = 131072
   D3DPRASTERCAPS_ANTIALIASEDGES = 4096
   D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT = 1024
   D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT = 2048
   D3DPRASTERCAPS_DITHER= 1
   D3DPRASTERCAPS_FOGRANGE = 65536
   D3DPRASTERCAPS_FOGTABLE = 256
   D3DPRASTERCAPS_FOGVERTEX = 128
   D3DPRASTERCAPS_MIPMAPLODBIAS = 8192
   D3DPRASTERCAPS_PAT = 8
   D3DPRASTERCAPS_ROP2 = 2
   D3DPRASTERCAPS_STIPPLE = 512
   D3DPRASTERCAPS_SUBPIXEL= 32
   D3DPRASTERCAPS_SUBPIXELX = 64
   D3DPRASTERCAPS_XOR = 4
   D3DPRASTERCAPS_ZBIAS = 16384
   D3DPRASTERCAPS_ZBUFFERLESSHSR = 32768
   D3DPRASTERCAPS_ZTEST = 16
End Enum
```

D3DPRASTERCAPS_ANISOTROPY
The device supports anisotropic filtering. For more information, see D3DRENDERSTATE_ANISOTROPY in the **CONST_D3DRENDERSTATETYPE** structure.

D3DPRASTERCAPS_ANTIALIASEDGES
The device can antialias lines forming the convex outline of objects. For more information, see D3DRENDERSTATE_EDGEANTIALIAS in the **CONST_D3DRENDERSTATETYPE** enumeration.

IDH__dx_CONST_D3DCAPSRASTER_d3d_vb

D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT
The device supports antialiasing that is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur. For more information, see the **CONST_D3DANTIALIASMODE** enumeration.

D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT
The device supports antialiasing that is not dependent on the sort order of the polygons. For more information, see the **CONST_D3DANTIALIASMODE** enumeration.

D3DPRASTERCAPS_DITHER
The device can dither to improve color resolution.

D3DPRASTERCAPS_FOGRANGE
The device supports range-based fog. In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene.

D3DPRASTERCAPS_FOGTABLE
The device calculates the fog value by referring to a lookup table containing fog values that are indexed to the depth of a given pixel.

D3DPRASTERCAPS_FOGVERTEX
The device calculates the fog value during the lighting operation, places the value into the alpha component given for the **specular** member of the **D3DTLVERTEX** structure, and interpolates the fog value during rasterization.

D3DPRASTERCAPS_MIPMAPLODBIAS
The device supports level-of-detail (LOD) bias adjustments. These bias adjustments enable an application to make a mipmap appear crisper or less sharp than it normally would. For more information about LOD bias in mipmaps, see D3DRENDERSTATE_MIPMAPLODBIAS**.**

D3DPRASTERCAPS_PAT
The driver can perform patterned drawing (lines or fills with the D3DRENDERSTATE_STIPPLEPATTERN render state) for the primitive being queried.

D3DPRASTERCAPS_ROP2
The device can support raster operations other than R2_COPYPEN.

D3DPRASTERCAPS_STIPPLE
The device can stipple polygons to simulate translucency.

D3DPRASTERCAPS_SUBPIXEL
The device performs subpixel placement of z, color, and texture data, rather than working with the nearest integer pixel coordinate. This helps avoid bleed-through due to z imprecision, and jitter of color and texture values for pixels. Note that there is no corresponding state that can be enabled and disabled; the device either performs subpixel placement or it does not, and this bit is present only so that the Direct3D client will be better able to determine what the rendering quality will be.

D3DPRASTERCAPS_SUBPIXELX

The device is subpixel accurate along the x-axis only and is clamped to an integer y-axis scan line. For information about subpixel accuracy, see D3DPRASTERCAPS_SUBPIXEL.

D3DPRASTERCAPS_TRANSLUCENTSORTINDEPENDENT
The device supports translucency that is not dependent on the sort order of the polygons. For more information, see the D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT.

D3DPRASTERCAPS_WBUFFER
The device supports depth buffering using w.

D3DPRASTERCAPS_WFOG
The device supports w-based fog. W-based fog is used when a perspective projection matrix is specified, but affine projections will still use z-based fog. The system considers a projection matrix that contains a non-zero value in the [3] [4] element to be a perspective projection matrix.

D3DPRASTERCAPS_XOR
The device can support **XOR** operations. If this flag is not set but D3DPRIM_RASTER_ROP2 is set, then **XOR** operations must still be supported.

D3DPRASTERCAPS_ZBIAS
The device supports z-bias values. These are integer values assigned to polygons that allow physically coplanar polygons to appear separate. For more information, see D3DRENDERSTATE_ZBIAS in the **CONST_D3DRENDERSTATETYPE** enumeration.

D3DPRASTERCAPS_ZBUFFERLESSHSR
The device can perform hidden-surface removal (HSR) without requiring the application to sort polygons, and without requiring the allocation of a depth-buffer. This leaves more video memory for textures. The method used to perform hidden-surface removal is hardware-dependent and is transparent to the application.

Z-bufferless HSR is performed if no depth-buffer surface is attached to the rendering-target surface and the depth-buffer comparison test is enabled (that is, when the state value associated with the D3DRENDERSTATE_ZENABLE enumeration constant is set to True).

D3DPRASTERCAPS_ZTEST
The device can perform z-test operations. This effectively renders a primitive and indicates whether any z pixels would have been rendered.

CONST_D3DCAPSSHADE

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DCAPSSHADE** enumeration defines polygon shading capability flags that are combined and present in the **lShadeCaps** member of the **D3DPRIMCAPS** type.

[#] IDH__dx_CONST_D3DCAPSSHADE_d3d_vb

```
Enum CONST_D3DCAPSSHADE
   D3DPSHADECAPS_ALPHAFLATBLEND = 4096
   D3DPSHADECAPS_ALPHAFLATSTIPPLED = 8192
   D3DPSHADECAPS_ALPHAGOURAUDBLEND = 16384
   D3DPSHADECAPS_ALPHAGOURAUDSTIPPLED = 32768
   D3DPSHADECAPS_ALPHAPHONGBLEND = 65536
   D3DPSHADECAPS_ALPHAPHONGSTIPPLED = 131072
   D3DPSHADECAPS_COLORFLATMONO = 1
   D3DPSHADECAPS_COLORFLATRGB = 2
   D3DPSHADECAPS_COLORGOURAUDMONO = 4
   D3DPSHADECAPS_COLORGOURAUDRGB = 8
   D3DPSHADECAPS_COLORPHONGMONO = 16
   D3DPSHADECAPS_COLORPHONGRGB = 32
   D3DPSHADECAPS_FOGFLAT = 262144
   D3DPSHADECAPS_FOGGOURAUD = 524288
   D3DPSHADECAPS_FOGPHONG = 1048576
   D3DPSHADECAPS_SPECULARFLATMONO = 64
   D3DPSHADECAPS_SPECULARFLATRGB = 128
   D3DPSHADECAPS_SPECULARGOURAUDMONO = 256
   D3DPSHADECAPS_SPECULARGOURAUDRGB = 512
   D3DPSHADECAPS_SPECULARPHONGMONO = 1024
   D3DPSHADECAPS_SPECULARPHONGRGB = 2048
End Enum
```

D3DPSHADECAPS_ALPHAFLATBLEND,
D3DPSHADECAPS_ALPHAFLATSTIPPLED

Device can support an alpha component for flat blended and stippled transparency, respectively (the D3DSHADE_FLAT state for the **CONST_D3DSHADEMODE** enumeration). In these modes, the alpha color component for a primitive is provided as part of the color for the first vertex of the primitive.

D3DPSHADECAPS_ALPHAGOURAUDBLEND,
D3DPSHADECAPS_ALPHAGOURAUDSTIPPLED

Device can support an alpha component for Gouraud blended and stippled transparency, respectively (the D3DSHADE_GOURAUD state for the **CONST_D3DSHADEMODE** enumeration). In these modes, the alpha color component for a primitive is provided at vertices and interpolated across a face along with the other color components.

D3DPSHADECAPS_ALPHAPHONGBLEND,
D3DPSHADECAPS_ALPHAPHONGSTIPPLED

Device can support an alpha component for Phong blended and stippled transparency, respectively (the D3DSHADE_PHONG state for the **CONST_D3DSHADEMODE** enumeration). In these modes, vertex parameters are reevaluated on a per-pixel basis, applying lighting effects for the red, green, and blue color components. Phong shading is not currently supported.

D3DPSHADECAPS_COLORFLATMONO,
D3DPSHADECAPS_COLORFLATRGB
Device can support colored flat shading in color models, respectively. In these modes, the color component for a primitive is provided as part of the color for the first vertex of the primitive. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORGOURAUDMONO,
D3DPSHADECAPS_COLORGOURAUDRGB
Device can support colored Gouraud shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, the color component for a primitive is provided at vertices and interpolated across a face along with the other color components. In monochromatic lighting modes, only the blue component of the color is interpolated; in RGB lighting modes, the red, green, and blue components are interpolated.

D3DPSHADECAPS_COLORPHONGMONO,
D3DPSHADECAPS_COLORPHONGRGB
Device can support colored Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. In these modes, vertex parameters are reevaluated on a per-pixel basis. Lighting effects are applied for the red, green, and blue color components in RGB mode, and for the blue component only for monochromatic mode. Phong shading is not currently supported.

D3DPSHADECAPS_FOGFLAT, D3DPSHADECAPS_FOGGOURAUD,
D3DPSHADECAPS_FOGPHONG
Device can support fog in the flat, Gouraud, and Phong shading models, respectively. Phong shading is not currently supported.

D3DPSHADECAPS_SPECULARFLATMONO,
D3DPSHADECAPS_SPECULARFLATRGB
Device can support specular highlights in flat shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively.

D3DPSHADECAPS_SPECULARGOURAUDMONO,
D3DPSHADECAPS_SPECULARGOURAUDRGB
Device can support specular highlights in Gouraud shading in color models, respectively.

D3DPSHADECAPS_SPECULARPHONGMONO,
D3DPSHADECAPS_SPECULARPHONGRGB
Device can support specular highlights in Phong shading in the **D3DCOLOR_MONO** and **D3DCOLOR_RGB** color models, respectively. Phong shading is not currently supported.

CONST_D3DCAPSTEXTURE

[This is preliminary documentation and subject to change.]

IDH__dx_CONST_D3DCAPSTEXTURE_d3d_vb

The **CONST_D3DCAPSTEXTURE** enumeration defines texturing capability flags that are combined and present in the **lTextureCaps** member of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSTEXTURE {
   D3DPTEXTURECAPS_ALPHA = 4
   D3DPTEXTURECAPS_BORDER = 16
   D3DPTEXTURECAPS_PERSPECTIVE = 1
   D3DPTEXTURECAPS_POW2 = 2
   D3DPTEXTURECAPS_SQUAREONLY = 32
   D3DPTEXTURECAPS_TRANSPARENCY = 8
End Enum
```

D3DPTEXTURECAPS_ALPHA
: Supports RGBA textures in the D3DTEX_DECAL and D3DTEX_MODULATE texture filtering modes. If this capability is not set, then only RGB textures are supported in those modes. Regardless of the setting of this flag, alpha must always be supported in D3DTEX_DECAL_MASK, D3DTEX_DECAL_ALPHA, and D3DTEX_MODULATE_ALPHA filtering modes whenever those filtering modes are available.

D3DPTEXTURECAPS_ALPHAPALETTE
: Supports palettized texture surfaces whose palettes contain alpha information (see DDPCAPS_ALPHA in the **DDCAPS** structure).

D3DPTEXTURECAPS_BORDER
: Superseded by D3DPTADDRESSCAPS_BORDER.

D3DPTEXTURECAPS_PERSPECTIVE
: Perspective correction is supported.

D3DPTEXTURECAPS_POW2
: All nonmipmapped textures must have widths and heights specified as powers of two if this flag is set. (Note that all mipmapped textures must always have dimensions that are powers of two.)

D3DPTEXTURECAPS_SQUAREONLY
: All textures must be square.

D3DPTEXTURECAPS_TEXREPEATNOTSCALEDBYSIZE
: Texture indices are not scaled by the texture size prior to interpolation.

D3DPTEXTURECAPS_TRANSPARENCY
: Texture transparency is supported. (Only those texels that are not the current transparent color are drawn.)

CONST_D3DCAPSTEXTUREADDRESS

[This is preliminary documentation and subject to change.]

The **CONST_D3DCAPSTEXTUREADDRESS** enumeration defines texture addressing capability flags that are combined and present in the **lTextureAddressCaps** member of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSTEXTUREADDRESS
   D3DPTADDRESSCAPS_BORDER = 8
   D3DPTADDRESSCAPS_CLAMP = 4
   D3DPTADDRESSCAPS_INDEPENDENTUV = 16
   D3DPTADDRESSCAPS_MIRROR = 2
   D3DPTADDRESSCAPS_WRAP = 1
End Enum
```

D3DPTADDRESSCAPS_BORDER
: Device supports setting coordinates outside the range [0.0, 1.0] to the border color, as specified by the D3DRENDERSTATE_BORDERCOLOR render state. This ability corresponds to the **D3DTADDRESS_BORDER** texture-addressing mode.

D3DPTADDRESSCAPS_CLAMP
: Device can clamp textures to addresses.

D3DPTADDRESSCAPS_INDEPENDENTUV
: Device can separate the texture-addressing modes of the u and v coordinates of the texture. This ability corresponds to the D3DRENDERSTATE_TEXTUREADDRESSU and D3DRENDERSTATE_TEXTUREADDRESSV render-state values.

D3DPTADDRESSCAPS_MIRROR
: Device can mirror textures to addresses.

D3DPTADDRESSCAPS_WRAP
: Device can wrap textures to addresses.

CONST_D3DCAPSTEXTUREBLEND

[This is preliminary documentation and subject to change.]

The **CONST_D3DCAPSTEXTUREBLEND** enumeration defines texture blending capability flags that are combined and present in the **lTextureBlendCaps** member of the **D3DPRIMCAPS** type.

IDH__dx_CONST_D3DCAPSTEXTUREADDRESS_d3d_vb
IDH__dx_CONST_D3DCAPSTEXTUREBLEND_d3d_vb

```
Enum CONST_D3DCAPSTEXTUREBLEND
   D3DPTBLENDCAPS_ADD = 128
   D3DPTBLENDCAPS_COPY = 64
   D3DPTBLENDCAPS_DECAL = 1
   D3DPTBLENDCAPS_DECALALPHA = 4
   D3DPTBLENDCAPS_DECALMASK = 16
   D3DPTBLENDCAPS_MODULATE = 2
   D3DPTBLENDCAPS_MODULATEALPHA = 8
   D3DPTBLENDCAPS_MODULATEMASK = 32
End Enum
```

D3DPTBLENDCAPS_ADD
: Supports the additive texture-blending mode, in which the Gouraud interpolants are added to the texture lookup with saturation semantics. This capability corresponds to the D3DTBLEND_ADD member of the **CONST_D3DTEXTUREBLEND** enumeration.

D3DPTBLENDCAPS_COPY
: Copy mode texture-blending (D3DTBLEND_COPY from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_DECAL
: Decal texture-blending mode (D3DTBLEND_DECAL from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_DECALALPHA
: Decal-alpha texture-blending mode (D3DTBLEND_DECALALPHA from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_DECALMASK
: Decal-mask texture-blending mode (D3DTBLEND_DECALMASK from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_MODULATE
: Modulate texture-blending mode (D3DTBLEND_MODULATE from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_MODULATEALPHA
: Modulate-alpha texture-blending mode (D3DTBLEND_MODULATEALPHA from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

D3DPTBLENDCAPS_MODULATEMASK
: Modulate-mask texture-blending mode (D3DTBLEND_MODULATEMASK from the **CONST_D3DTEXTUREBLEND** enumeration) is supported.

CONST_D3DCAPSTEXTUREFILTER

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_CONST_D3DCAPSTEXTUREFILTER_d3d_vb

The **CONST_D3DCAPSTEXTUREFILTER** enumeration defines texture filtering capability flags that are combined and present in the **lTextureFilterCaps** of the **D3DPRIMCAPS** type.

```
Enum CONST_D3DCAPSTEXTUREFILTER
   D3DPTFILTERCAPS_LINEAR = 2
   D3DPTFILTERCAPS_LINEARMIPLINEAR = 32
   D3DPTFILTERCAPS_LINEARMIPNEAREST = 16
   D3DPTFILTERCAPS_MIPLINEAR = 8
   D3DPTFILTERCAPS_MIPNEAREST = 4
   D3DPTFILTERCAPS_NEAREST = 1
End Enum
```

D3DPTFILTERCAPS_LINEAR
: A weighted average of a 2×2 area of texels surrounding the desired pixel is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DPTFILTERCAPS_LINEARMIPLINEAR
: Similar to D3DPTFILTERCAPS_MIPLINEAR, but interpolates between the two nearest mipmaps.

D3DPTFILTERCAPS_LINEARMIPNEAREST
: The mipmap chosen is the mipmap whose texels most closely match the size of the pixel to be textured. The D3DFILTER_LINEAR method is then used with the texture.

D3DPTFILTERCAPS_MIPLINEAR
: Two mipmaps are chosen whose texels most closely match the size of the pixel to be textured. The D3DFILTER_NEAREST method is then used with each texture to produce two values which are then weighted to produce a final texel value.

D3DPTFILTERCAPS_MIPNEAREST
: Similar to D3DPTFILTERCAPS_NEAREST, but uses the appropriate mipmap for texel selection.

D3DPTFILTERCAPS_NEAREST
: The texel with coordinates nearest to the desired pixel value is used. This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

CONST_D3DCLEARFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DCLEARFLAGS** enumeration defines flags that are used to determine the behavior of the **Direct3DViewport3.Clear** and **Direct3DViewport3.Clear2** methods.

```
Enum CONST_D3DCLEARFLAGS
```

[#] IDH__dx_CONST_D3DCLEARFLAGS_d3d_vb

```
    D3DCLEAR_ALL = 7
    D3DCLEAR_STENCIL = 4 ' Not supported by 3DViewport.Clear
    D3DCLEAR_TARGET = 1
    D3DCLEAR_ZBUFFER = 2
  End Enum
```

D3DCLEAR_ALL
: For the **D3DViewport3.Clear2** method, clear the rendering target, stencil buffer, and depth-buffer surfaces. For the For the **D3DViewport3.Clear** method, clear the rendering target and depth-buffer surfaces.

D3DCLEAR_STENCIL
: Clear the stencil buffer to the value in the *dwStencil* parameter. (This flag is not supported by the **D3DViewport3.Clear** method.)

D3DCLEAR_TARGET
: For the **D3DViewport3.Clear2** method, clear the rendering target to the color in the *dwColor* parameter; for the **D3DViewport3.Clear** method, clear the rendering target to the color of the background material.

D3DCLEAR_ZBUFFER
: For the **D3DViewport3.Clear2** method, clear the depth-buffer to the value in the *dvZ* parameter; for the **D3DViewport3.Clear** method, clear the depth buffer to the default value.

CONST_D3DCLIPFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DCLIPFLAGS** enumeration defines clipping flags used in the **D3DSTATUS** and **D3DTRANSFORMDATA** types.

```
  Enum CONST_D3DCLIPFLAGS
    ' Combination and general flags
    D3DSTATUS_CLIPINTERSECTIONALL = 17891328
    D3DSTATUS_CLIPUNIONALL = 17891328
    D3DSTATUS_DEFAULT = 34668544
    D3DSTATUS_ZNOTVISIBLE = 16777216
    ' Clip intersection flags
    D3DSTATUS_CLIPINTERSECTIONBACK = 131072
    D3DSTATUS_CLIPINTERSECTIONBOTTOM = 32768
    D3DSTATUS_CLIPINTERSECTIONFRONT = 65536
    D3DSTATUS_CLIPINTERSECTIONGEN0 = 262144
    D3DSTATUS_CLIPINTERSECTIONGEN1 = 524288
    D3DSTATUS_CLIPINTERSECTIONGEN2 = 1048576
    D3DSTATUS_CLIPINTERSECTIONGEN3 = 2097152
    D3DSTATUS_CLIPINTERSECTIONGEN4 = 4194304
    D3DSTATUS_CLIPINTERSECTIONGEN5 = 8388608
```

[#] IDH__dx_CONST_D3DCLIPFLAGS_d3d_vb

```
    D3DSTATUS_CLIPINTERSECTIONLEFT = 4096
    D3DSTATUS_CLIPINTERSECTIONRIGHT = 8192
    D3DSTATUS_CLIPINTERSECTIONTOP = 16384
    ' Clip union flags
    D3DSTATUS_CLIPUNIONBACK = 32
    D3DSTATUS_CLIPUNIONBOTTOM = 8
    D3DSTATUS_CLIPUNIONFRONT = 16
    D3DSTATUS_CLIPUNIONGEN0 = 64
    D3DSTATUS_CLIPUNIONGEN1 = 128
    D3DSTATUS_CLIPUNIONGEN2 = 256
    D3DSTATUS_CLIPUNIONGEN3 = 512
    D3DSTATUS_CLIPUNIONGEN4 = 1024
    D3DSTATUS_CLIPUNIONGEN5 = 2048
    D3DSTATUS_CLIPUNIONLEFT = 1
    D3DSTATUS_CLIPUNIONRIGHT = 2
    D3DSTATUS_CLIPUNIONTOP = 4
    ' Basic clipping flags
    D3DCLIP_BACK = 32
    D3DCLIP_BOTTOM = 8
    D3DCLIP_FRONT = 16
    D3DCLIP_GEN0 = 64
    D3DCLIP_GEN1 = 128
    D3DCLIP_GEN2 = 256
    D3DCLIP_GEN3 = 512
    D3DCLIP_GEN4 = 1024
    D3DCLIP_GEN5 = 2048
    D3DCLIP_LEFT = 1
    D3DCLIP_RIGHT = 2
    D3DCLIP_TOP = 4
End Enum
```

Combination and General Flags

D3DSTATUS_CLIPINTERSECTIONALL
: Combination of all CLIPINTERSECTION flags.

D3DSTATUS_CLIPUNIONALL
: Combination of all CLIPUNION flags.

D3DSTATUS_DEFAULT
: Combination of D3DSTATUS_CLIPINTERSECTIONALL and D3DSTATUS_ZNOTVISIBLE flags. This value is the default.

D3DSTATUS_ZNOTVISIBLE
: Indicates that the rendered primitive is not visible. This flag is set or cleared by the system when rendering with z-checking enabled (see D3DRENDERSTATE_ZVISIBLE).

Clip Intersection Flags

D3DSTATUS_CLIPINTERSECTIONBACK

Logical **AND** of the clip flags for the vertices compared to the back clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONBOTTOM
Logical **AND** of the clip flags for the vertices compared to the bottom of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONFRONT
Logical **AND** of the clip flags for the vertices compared to the front clipping plane of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONGEN0 through
D3DSTATUS_CLIPINTERSECTIONGEN5
Logical **AND** of the clip flags for application-defined clipping planes.

D3DSTATUS_CLIPINTERSECTIONLEFT
Logical **AND** of the clip flags for the vertices compared to the left side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONRIGHT
Logical **AND** of the clip flags for the vertices compared to the right side of the viewing frustum.

D3DSTATUS_CLIPINTERSECTIONTOP
Logical **AND** of the clip flags for the vertices compared to the top of the viewing frustum.

Clip Union Flags

D3DSTATUS_CLIPUNIONBACK
Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM
Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT
Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5
Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT
Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT
Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP
Equal to D3DCLIP_TOP.

Basic Clipping Flags

D3DCLIP_BACK
All vertices are clipped by the back plane of the viewing frustum.

D3DCLIP_BOTTOM
All vertices are clipped by the bottom plane of the viewing frustum.

D3DCLIP_FRONT
All vertices are clipped by the front plane of the viewing frustum.

D3DCLIP_LEFT

All vertices are clipped by the left plane of the viewing frustum.

D3DCLIP_RIGHT
All vertices are clipped by the right plane of the viewing frustum.

D3DCLIP_TOP
All vertices are clipped by the top plane of the viewing frustum.

D3DCLIP_GEN0 through D3DCLIP_GEN5
Application-defined clipping planes.

CONST_D3DCLIPSTATUSFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DCLIPSTATUSFLAGS** enumeration defines flags that are used in the **D3DCLIPSTATUS** type.

```
Enum CONST_D3DCLIPSTATUSFLAGS
   D3DCLIPSTATUS_EXTENTS2 = 2
   D3DCLIPSTATUS_EXTENTS3 = 4
   D3DCLIPSTATUS_STATUS = 1
End Enum
```

D3DCLIPSTATUS_EXTENTS2
The structure describes the current 2-D extents. This flag cannot be combined with D3DCLIPSTATUS_EXTENTS3.

D3DCLIPSTATUS_EXTENTS3
Not currently implemented.

D3DCLIPSTATUS_STATUS
The structure describes the current clip status.

CONST_D3DCMPFUNC

[This is preliminary documentation and subject to change.]

The **CONST_D3DCMPFUNC** enumeration defines the supported compare functions for the D3DRENDERSTATE_ZFUNC, D3DRENDERSTATE_ALPHAFUNC, and D3DRENDERSTATE_STENCILFUNC render states.

```
Enum CONST_D3DCMPFUNC
   D3DCMP_NEVER       = 1
   D3DCMP_LESS        = 2
   D3DCMP_EQUAL       = 3
   D3DCMP_LESSEQUAL   = 4
   D3DCMP_GREATER     = 5
   D3DCMP_NOTEQUAL    = 6
```

IDH__dx_CONST_D3DCLIPSTATUSFLAGS_d3d_vb
IDH__dx_CONST_D3DCMPFUNC_d3d_vb

```
    D3DCMP_GREATEREQUAL = 7
    D3DCMP_ALWAYS       = 8
    D3DCMP_FORCE_DWORD  = 0x7fffffff
  End Enum
```

D3DCMP_NEVER
Always fail the test.

D3DCMP_LESS
Accept the new pixel if its value is less than the value of the current pixel.

D3DCMP_EQUAL
Accept the new pixel if its value equals the value of the current pixel.

D3DCMP_LESSEQUAL
Accept the new pixel if its value is less than or equal to the value of the current pixel.

D3DCMP_GREATER
Accept the new pixel if its value is greater than the value of the current pixel.

D3DCMP_NOTEQUAL
Accept the new pixel if its value does not equal the value of the current pixel.

D3DCMP_GREATEREQUAL
Accept the new pixel if its value is greater than or equal to the value of the current pixel.

D3DCMP_ALWAYS
Always pass the test.

D3DCMP_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

CONST_D3DCOLORMODEL

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DCOLORMODEL** enumeration is used to define the color model in which the system will run. A driver can expose either or both flags in the **CONST_COLORMODEL** member of the **D3DDEVICEDESC** structure.

```
  Enum CONST_D3DCOLORMODEL
    D3DCOLOR_MONO = 1
    D3DCOLOR_RGB = 2
  End Enum
```

D3DCOLOR_MONO
Use a monochromatic model (or ramp model). In this model, the blue component of a vertex color is used to define the brightness of a lit vertex.

D3DCOLOR_RGB

[#] IDH__dx_CONST_D3DCOLORMODEL_d3d_vb

Use a full RGB model.

CONST_D3DCULL

[This is preliminary documentation and subject to change.]

The **CONST_D3DCULL** enumeration defines the supported cull modes. These define how back faces are culled when rendering a geometry.

```
Enum CONST_D3DCULL
    D3DCULL_NONE = 1
    D3DCULL_CW   = 2
    D3DCULL_CCW  = 3
    D3DCULL_FORCE_DWORD   = 0x7fffffff
End Enum
```

D3DCULL_NONE
Do not cull back faces.

D3DCULL_CW
Cull back faces with clockwise vertices.

D3DCULL_CCW
Cull back faces with counterclockwise vertices.

D3DCULL_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

See Also

D3DPRIMCAPS, **CONST_D3DRENDERSTATETYPE**

CONST_D3DDEVICEDESCCAPS

[This is preliminary documentation and subject to change.]

The **CONST_D3DDEVICEDESCCAPS** enumeration defines device capability flags that are combined and present in the **lDevCaps** member of the **D3DDEVICEDESC** type.

```
Enum CONST_D3DDEVICEDESCCAPS
    D3DDEVCAPS_CANRENDERAFTERFLIP = 2048
    D3DDEVCAPS_DRAWPRIMTLVERTEX = 1024
    D3DDEVCAPS_EXECUTESYSTEMMEMORY = 16
    D3DDEVCAPS_EXECUTEVIDEOMEMORY = 32
    D3DDEVCAPS_FLOATTLVERTEX = 1
    D3DDEVCAPS_SORTDECREASINGZ = 4
```

IDH__dx_CONST_D3DCULL_d3d_vb
IDH__dx_CONST_D3DDEVICEDESCCAPS_d3d_vb

```
   D3DDEVCAPS_SORTEXACT = 8
   D3DDEVCAPS_SORTINCREASINGZ = 2
   D3DDEVCAPS_TEXTURENONLOCALVIDMEM = 4096
   D3DDEVCAPS_TEXTURESYSTEMMEMORY = 256
   D3DDEVCAPS_TEXTUREVIDEOMEMORY = 512
   D3DDEVCAPS_TLVERTEXSYSTEMMEMORY = 64
   D3DDEVCAPS_TLVERTEXVIDEOMEMORY = 128
End Enum
```

D3DDEVCAPS_CANRENDERAFTERFLIP
: Device can queue rendering commands after a page flip. Applications should not change their behavior if this flag is set; this capability simply means that the device is relatively fast.

D3DDEVCAPS_DRAWPRIMTLVERTEX
: Device exports a DrawPrimitive-aware HAL.

D3DDEVCAPS_EXECUTESYSTEMMEMORY
: Device can use execute buffers from system memory.

D3DDEVCAPS_EXECUTEVIDEOMEMORY
: Device can use execute buffer from video memory.

D3DDEVCAPS_FLOATTLVERTEX
: Device accepts floating point for post-transform vertex data.

D3DDEVCAPS_SORTDECREASINGZ
: Device needs data sorted for decreasing depth.

D3DDEVCAPS_SORTEXACT
: Device needs data sorted exactly.

D3DDEVCAPS_SORTINCREASINGZ
: Device needs data sorted for increasing depth.

D3DDEVCAPS_TEXREPEATNOTSCALEDBYSIZE
: Device defers scaling of texture indices by the texture size until after the texture address mode is applied.

D3DDEVCAPS_TEXTURENONLOCALVIDMEM
: Device can retrieve textures from non-local video (AGP) memory.

D3DDEVCAPS_TEXTURESYSTEMMEMORY
: Device can retrieve textures from system memory.

D3DDEVCAPS_TEXTUREVIDEOMEMORY
: Device can retrieve textures from device memory.

D3DDEVCAPS_TLVERTEXSYSTEMMEMORY
: Device can use buffers from system memory for transformed and lit vertices.

D3DDEVCAPS_TLVERTEXVIDEOMEMORY
: Device can use buffers from video memory for transformed and lit vertices.

CONST_D3DDEVICEDESCFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DDEVICEDESCFLAGS** enumeration defines flags that are used in the **lFlags** member of the **D3DDEVICEDESC** type.

```
Enum CONST_D3DDEVICEDESCFLAGS
    D3DDD_BCLIPPING = 16
    D3DDD_COLORMODEL = 1
    D3DDD_DEVCAPS = 2
    D3DDD_DEVICERENDERBITDEPTH = 128
    D3DDD_DEVICEZBUFFERBITDEPTH = 256
    D3DDD_LIGHTINGCAPS = 8
    D3DDD_LINECAPS = 32
    D3DDD_MAXBUFFERSIZE = 512
    D3DDD_MAXVERTEXCOUNT = 1024
    D3DDD_TRANSFORMCAPS = 4
    D3DDD_TRICAPS = 64
End Enum
```

D3DDD_BCLIPPING
: The **lClipping** member of the **D3DDEVICEDESC** type is valid.

D3DDD_COLORMODEL
: The **lColorModel** member of the **D3DDEVICEDESC** type is valid.

D3DDD_DEVCAPS
: The **lDevCaps** member of the **D3DDEVICEDESC** type is valid.

D3DDD_DEVICERENDERBITDEPTH
: The **lDeviceRenderBitDepth** member of the **D3DDEVICEDESC** type is valid.

D3DDD_DEVICEZBUFFERBITDEPTH
: The **lDeviceZBufferBitDepth** member of the **D3DDEVICEDESC** type is valid.

D3DDD_LIGHTINGCAPS
: The **dlcLightingCaps** member of the **D3DDEVICEDESC** type is valid.

D3DDD_LINECAPS
: The **dpcLineCaps** member of the **D3DDEVICEDESC** type is valid.

D3DDD_MAXBUFFERSIZE
: The **lMaxBufferSize** member of the **D3DDEVICEDESC** type is valid.

D3DDD_MAXVERTEXCOUNT
: The **lMaxVertexCount** member of the **D3DDEVICEDESC** type is valid.

D3DDD_TRANSFORMCAPS
: The **lTransformCaps** member of the **D3DDEVICEDESC** type is valid.

D3DDD_TRICAPS
: The **dpcTriCaps** member of the **D3DDEVICEDESC** type is valid.

IDH__dx_CONST_D3DDEVICEDESCFLAGS_d3d_vb

CONST_D3DDPFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DDPFLAGS** enumeration defines flags that are used to determine the behavior of the Direct3D rendering methods. These flags are used with the **Direct3DDevice3.DrawPrimitive**, **Direct3DDevice3.DrawIndexedPrimitive**, **Direct3DDevice3.DrawPrimitiveVB**, **Direct3DDevice3.DrawIndexedPrimitiveVB**, **Direct3DDevice3.Begin**, and **Direct3DDevice3.BeginIndexed**.

```
Enum CONST_D3DDPFLAGS
   D3DDP_DEFAULT = 0
   D3DDP_DONOTCLIP = 4
   D3DDP_DONOTLIGHT = 16
   D3DDP_DONOTUPDATEEXTENTS = 8
   D3DDP_WAIT = 1
End Enum
```

D3DDP_DEFAULT
: Perform normal rendering. (Return as soon as the polygons are sent to the card.)

D3DDP_DONOTCLIP
: The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)

D3DDP_DONOTLIGHT
: Disables the Direct3D lighting engine. The system uses the diffuse and specular components at each vertex for shading when it rasterizes the set of primitives. If a diffuse or specular component is not specified, the system uses the default color for the missing component (0xFFFFFFFF for diffuse and 0x00000000 for specular).

D3DDP_DONOTUPDATEEXTENTS
: Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice3.GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_WAIT
: Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

[#] IDH__dx_CONST_D3DDPFLAGS_d3d_vb

CONST_D3DFILLMODE

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DFILLMODE** enumeration contains constants describing the fill mode. These values are used by the D3DRENDERSTATE_FILLMODE render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

```
Enum CONST_D3DFILLMODE
   D3DFILL_POINT     = 1
   D3DFILL_WIREFRAME = 2
   D3DFILL_SOLID     = 3
   D3DFILL_FORCE_DWORD   = 0x7fffffff
End Enum
```

D3DFILL_POINT
: Fill points.

D3DFILL_WIREFRAME
: Fill wireframes. This fill mode currently does not work for clipped primitives when you are using the DrawPrimitive methods.

D3DFILL_SOLID
: Fill solids.

D3DFILL_FORCE_DWORD
: Forces this enumeration to be 32 bits in size.

CONST_D3DFINDDEVICESEARCHFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DFINDDEVICESEARCHFLAGS** enumeration contains flags defining the type of device the application wants to find. This member can be one or more of the following values:

```
Enum CONST_D3DFINDDEVICESEARCHFLAGS
   D3DFDS_ALPHACMPCAPS = 256 (&H100)
   D3DFDS_COLORMODEL = 1
   D3DFDS_DSTBLENDCAPS = 1024 (&H400)
   D3DFDS_GUID = 2
   D3DFDS_HARDWARE = 4
   D3DFDS_LINES = 16 (&H10)
   D3DFDS_MISCCAPS = 32 (&H20)
   D3DFDS_RASTERCAPS = 64 (&H40)
```

IDH__dx_CONST_D3DFILLMODE_d3d_vb
IDH__dx_CONST_D3DFINDDEVICESEARCHFLAGS_d3d_vb

```
    D3DFDS_SHADECAPS = 2048 (&H800)
    D3DFDS_SRCBLENDCAPS = 512 (&H200)
    D3DFDS_TEXTUREADDRESSCAPS = 32768 (&H8000)
    D3DFDS_TEXTUREBLENDCAPS = 16384 (&H4000)
    D3DFDS_TEXTURECAPS = 4096 (&H1000)
    D3DFDS_TEXTUREFILTERCAPS = 8192 (&H2000)
    D3DFDS_TRIANGLES = 8
    D3DFDS_ZCMPCAPS = 128 (&H80)
End Enum
```

D3DFDS_ALPHACMPCAPS
: Match the **lAlphaCmpCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

D3DFDS_COLORMODEL
: Match the color model specified in the **dcmColorModel** member of this type.

D3DFDS_DSTBLENDCAPS
: Match the **lDestBlendCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

D3DFDS_GUID
: Match the globally unique identifier (GUID) specified in the **guid** member of this type.

D3DFDS_HARDWARE
: Match the hardware or software search specification given in the **bHardware** member of this type.

D3DFDS_LINES
: Match the **D3DPRIMCAPS** type specified by the **dpcLineCaps** member of the **D3DDEVICEDESC** type.

D3DFDS_MISCCAPS
: Match the **lMiscCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

D3DFDS_RASTERCAPS
: Match the **lRasterCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

D3DFDS_SHADECAPS
: Match the **lShadeCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

D3DFDS_SRCBLENDCAPS
: Match the **lSrcBlendCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

D3DFDS_TEXTUREBLENDCAPS
: Match the **lTextureBlendCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

D3DFDS_TEXTURECAPS

Match the **lTextureCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

D3DFDS_TEXTUREFILTERCAPS
Match the **lTextureFilterCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

D3DFDS_TRIANGLES
Match the **D3DPRIMCAPS** type specified by the **dpcTriCaps** member of the **D3DDEVICEDESC** type.

D3DFDS_ZCMPCAPS
Match the **lZCmpCaps** member of the **D3DPRIMCAPS** type specified as the **dpcPrimCaps** member of this type.

CONST_D3DFOGMODE

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DFOGMODE** enumeration contains constants describing the fog mode. These values are used by the D3DRENDERSTATE_FOGTABLEMODE render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

```
Enum CONST_D3DFOGMODE
   D3DFOG_NONE   = 0
   D3DFOG_EXP    = 1
   D3DFOG_EXP2   = 2
   D3DFOG_LINEAR = 3
   D3DFOG_FORCE_DWORD   = 0x7fffffff
End Enum
```

D3DFOG_NONE
No fog effect.

D3DFOG_EXP
The fog effect intensifies exponentially, according to the following formula:

$$f = 1 / e^{d \times density}$$

D3DFOG_EXP2
The fog effect intensifies exponentially with the square of the distance, according to the following formula:

$$f = 1 / e^{(d \times density)^2}$$

D3DFOG_LINEAR
The fog effect intensifies linearly between the start and end points, according to the following formula:

[#] IDH__dx_CONST_D3DFOGMODE_d3d_vb

$$f = \frac{end - a}{end - start}$$

This is the only fog mode currently supported.

D3DFOG_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

Remarks

In monochromatic (ramp) lighting mode, fog works properly only when the fog color is black. (If there is no lighting, any fog color will work, since in this case any fog color is effectively black.)

Note

Fog can be considered a measure of visibility — the lower the fog value produced by one of the fog equations, the less visible an object is.

CONST_D3DFVFCAPSFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DFVFCAPSFLAGS** enumeration defines values present within or used with the **lFVFCaps** member of the **D3DDEVICEDESC** type.

```
Enum CONST_D3DFVFCAPSFLAGS
   D3DFVFCAPS_DONOTSTRIPELEMENTS = 524288
End Enum
```

D3DFVFCAPS_DONOTSTRIPELEMENTS
Device prefers that vertex elements not be stripped. That is, if the vertex format contains elements that will not be used with the current render states, there is no need to regenerate the vertices. If this capability flag is not present, stripping extraneous elements from the vertex format will provide better performance.

CONST_D3DFVFFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DFVFFLAGS** enumeration defines flexible vertex format flags for use with the DrawPrimitive rendering methods. For details, see Flexible Vertex Format Flags.

IDH__dx_CONST_D3DFVFCAPSFLAGS_d3d_vb
IDH__dx_CONST_D3DFVFFLAGS_d3d_vb

CONST_D3DIMERR

[This is preliminary documentation and subject to change.]

The **CONST_D3DIMERR** enumeration defines error codes raised by the system, and are not otherwise useful. For descriptions of these error codes, see Error Codes.

CONST_D3DLIGHT2FLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DLIGHT2FLAGS** enumeration defines flags that can be combined in the **lFlags** member of the **D3DLIGHT2** type.

```
Enum CONST_D3DLIGHT2FLAGS
   D3DLIGHT_ACTIVE = 1
   D3DLIGHT_NO_SPECULAR = 2
End Enum
```

D3DLIGHT_ACTIVE
: Enables the light. This flag must be set to enable the light; if it is not set, the light is ignored.

D3DLIGHT_NO_SPECULAR
: Turns off specular highlights for the light.

CONST_D3DLIGHTCAPSFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DLIGHTCAPSFLAGS** enumeration defines lighting capability flags that are combined and present in the **lCaps** member of the **D3DLIGHTINGCAPS** type.

```
Enum CONST_D3DLIGHTCAPSFLAGS
   D3DLIGHTCAPS_DIRECTIONAL = 4
   D3DLIGHTCAPS_POINT = 1
   D3DLIGHTCAPS_SPOT = 2
   D3DLIGHTCAPS_PARALLELPOINT    =  8
End Enum
```

D3DLIGHTCAPS_DIRECTIONAL
: Supports directional lights.

D3DLIGHTCAPS_PARALLELPOINT
: Supports parallel point lights.

IDH__dx_CONST_D3DIMERR_d3d_vb
IDH__dx_CONST_D3DLIGHT2FLAGS_d3d_vb
IDH__dx_CONST_D3DLIGHTCAPSFLAGS_d3d_vb

D3DLIGHTCAPS_POINT
Supports point lights.

D3DLIGHTCAPS_SPOT
Supports spotlights.

CONST_D3DLIGHTINGMODELFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DLIGHTINGMODELFLAGS** enumeration defines lighting model capability flags that are combined and present in the **lLightingModel** member of the **D3DLIGHTINGCAPS** type.

```
Enum CONST_D3DLIGHTINGMODELFLAGS
   D3DLIGHTINGMODEL_MONO = 2
   D3DLIGHTINGMODEL_RGB = 1
End Enum
```

D3DLIGHTINGMODEL_MONO
Monochromatic lighting model.

D3DLIGHTINGMODEL_RGB
RGB lighting model.

CONST_D3DLIGHTSTATETYPE

[This is preliminary documentation and subject to change.]

The **CONST_D3DLIGHTSTATETYPE** enumeration defines the light state for the **Direct3DDevice3.SetLightState** method.

```
Enum CONST_D3DLIGHTSTATETYPE
   D3DLIGHTSTATE_MATERIAL   = 1
   D3DLIGHTSTATE_AMBIENT    = 2
   D3DLIGHTSTATE_COLORMODEL = 3
   D3DLIGHTSTATE_FOGMODE    = 4
   D3DLIGHTSTATE_FOGSTART   = 5
   D3DLIGHTSTATE_FOGEND     = 6
   D3DLIGHTSTATE_FOGDENSITY = 7
   D3DLIGHTSTATE_COLORVERTEX      = 8
   D3DLIGHTSTATE_FORCE_DWORD     = 0x7fffffff
End Enum
```

D3DLIGHTSTATE_MATERIAL

IDH__dx_CONST_D3DLIGHTINGMODELFLAGS_d3d_vb
IDH__dx_CONST_D3DLIGHTSTATETYPE_d3d_vb

Defines the material that is lit and used to compute the final color and intensity values during rasterization. The default value is Nothing. This value must be set when you use textures in ramp mode.

D3DLIGHTSTATE_AMBIENT
Sets the color and intensity of the current ambient light. If an application specifies this value, it should not specify a light as a parameter. The default value is 0.

D3DLIGHTSTATE_COLORMODEL
Either D3DCOLOR_MONO or D3DCOLOR_RGB constant. The default value is D3DCOLOR_RGB.

D3DLIGHTSTATE_FOGMODE
One of the constants of the **CONST_D3DFOGMODE** enumeration. The default value is D3DFOG_NONE.

D3DLIGHTSTATE_FOGSTART
Defines the starting value for fog. The default value is 1.0.

D3DLIGHTSTATE_FOGEND
Defines the ending value for fog. The default value is 100.0.

D3DLIGHTSTATE_FOGDENSITY
Defines the density setting for fog. The default value is 1.0.

D3DLIGHTSTATE_COLORVERTEX
Enables or disables the use of the vertex color in lighting calculations for vertices whose vertex format (specified as a flexible vertex format) includes color information. The default value, True, enables the use of the vertex color in lighting. Set this to False to cause the system to ignore the vertex color. Per-vertex color is supported only by lights for which properties are defined by a **D3DLIGHT2** structure.

D3DLIGHTSTATE_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

Remarks

Setting D3DLIGHTSTATE_COLORVERTEX to False instructs the geometry pipeline to ignore part of each vertex (the vertex color). The only reason to use this light state is to change the appearance of the geometry without respecifying it in a different vertex format.

If D3DLIGHTSTATE_COLORVERTEX is set to True and a diffuse vertex color is present, the output alpha is equal to the diffuse alpha for the vertex. Otherwise, output alpha is equal to the alpha component of diffuse material, clamped to the range [0, 255].

You can disable or enable lighting by including or omitting the D3DDP_DONOTLIGHT flag when calling a standard **Direct3DDevice3** rendering method, such as **Direct3DDevice3.DrawPrimitive**. If you are using vertex buffers, disable or enable lighting by ommiting or including the D3DVOP_LIGHT flag when you call the **Direct3DVertexBuffer.ProcessVertices** method.

See Also

Light Properties

CONST_D3DLIGHTTYPE

[This is preliminary documentation and subject to change.]

The **CONST_D3DLIGHTTYPE** enumeration defines flags that identify light types. These flags are used in the **lType** member of the **D3DLIGHT2** type.

```
Enum CONST_D3DLIGHTTYPE
   D3DLIGHT_DIRECTIONAL = 3
   D3DLIGHT_PARALLELPOINT = 4
   D3DLIGHT_POINT = 1
   D3DLIGHT_SPOT = 2
End Enum
```

D3DLIGHT_DIRECTIONAL
: Light is a directional source. This is equivalent to using a point light source at an infinite distance.

D3DLIGHT_PARALLELPOINT
: Light is a parallel point source. This light type acts like a directional light except its direction is the vector going from the light position to the origin of the geometry it is illuminating.

D3DLIGHT_POINT
: Light is a point source. The light has a position in space and radiates light in all directions.

D3DLIGHT_SPOT
: Light is a spotlight source. This light is something like a point light except that the illumination is limited to a cone. This light type has a direction and several other parameters which determine the shape of the cone it produces. For information about these parameters, see the **D3DLIGHT2** type.

CONST_D3DNEXTFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DNEXTFLAGS** defines behavior flags that can be used in the *flags* parameter of the **Direct3DDevice3.NextViewport** method.

```
Enum CONST_D3DNEXTFLAGS
   D3DNEXT_HEAD = 2
   D3DNEXT_NEXT = 1
   D3DNEXT_TAIL = 4
```

IDH__dx_CONST_D3DLIGHTTYPE_d3d_vb
IDH__dx_CONST_D3DNEXTFLAGS_d3d_vb

```
End Enum
```

D3DNEXT_HEAD
Retrieve the item at the beginning of the list.

D3DNEXT_NEXT
Retrieve the next item in the list.

D3DNEXT_TAIL
Retrieve the item at the end of the list.

CONST_D3DPALFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DPALFLAGS** enumeration defines palette entry flags that can be combined and used in the **flags** member of the **PALETTEENTRY** type.

```
Enum CONST_D3DPALFLAGS
   D3DPAL_FREE = 0
   D3DPAL_READONLY = 64
   D3DPAL_RESERVED = 128
End Enum
```

D3DPAL_FREE

D3DPAL_READONLY

D3DPAL_RESERVED

CONST_D3DPRIMITIVETYPE

[This is preliminary documentation and subject to change.]

The **CONST_D3DPRIMITIVETYPE** enumeration lists the primitives supported by DrawPrimitive methods.

```
Enum CONST_D3DPRIMITIVETYPE
   D3DPT_POINTLIST     = 1
   D3DPT_LINELIST      = 2
   D3DPT_LINESTRIP     = 3
   D3DPT_TRIANGLELIST  = 4
   D3DPT_TRIANGLESTRIP = 5
   D3DPT_TRIANGLEFAN   = 6
   D3DPT_FORCE_DWORD   = 0x7fffffff
```

IDH__dx_CONST_D3DPALFLAGS_d3d_vb
IDH__dx_CONST_D3DPRIMITIVETYPE_d3d_vb

End Enum

D3DPT_POINTLIST
Renders the vertices as a collection of isolated points.

D3DPT_LINELIST
Renders the vertices as a list of isolated straight line segments. Calls using this primitive type will fail if the count is less than 2, or is odd.

D3DPT_LINESTRIP
Renders the vertices as a single polyline. Calls using this primitive type will fail if the count is less than 2.

D3DPT_TRIANGLELIST
Renders the specified vertices as a sequence of isolated triangles. Each group of 3 vertices defines a separate triangle. Calls using this primitive type will fail if the count is less than 3, or if not evenly divisible by 3.

Backface culling is affected by the current winding order render state.

D3DPT_TRIANGLESTRIP
Renders the vertices as a triangle strip. Calls using this primitive type will fail if the count is less than 3. The backface removal flag is automatically flipped on even numbered triangles.

D3DPT_TRIANGLEFAN
Renders the vertices as a triangle fan. Calls using this primitive type will fail if the count is less than 3.

D3DPT_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

Remarks

Using triangle strips or fans is often more efficient than using triangle lists, as fewer vertices are duplicated.

See Also

Direct3DDevice3.Begin, **Direct3DDevice3.BeginIndexed**, **Direct3DDevice3.DrawIndexedPrimitive**, **Direct3DDevice3.DrawPrimitive**

CONST_D3DRENDERSTATETYPE

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DRENDERSTATETYPE** enumeration describes the render state for the D3DOP_STATERENDER opcode. The values mentioned in the following descriptions are set in the second member of this structure.

Enum CONST_D3DRENDERSTATETYPE

[#] IDH__dx_CONST_D3DRENDERSTATETYPE_d3d_vb

```
D3DRENDERSTATE_TEXTUREHANDLE      = 1,
D3DRENDERSTATE_ANTIALIAS          = 2,
D3DRENDERSTATE_TEXTUREADDRESS     = 3,
D3DRENDERSTATE_TEXTUREPERSPECTIVE = 4,
D3DRENDERSTATE_WRAPU              = 5,
D3DRENDERSTATE_WRAPV              = 6,
D3DRENDERSTATE_ZENABLE            = 7,
D3DRENDERSTATE_FILLMODE           = 8,
D3DRENDERSTATE_SHADEMODE          = 9,
D3DRENDERSTATE_MONOENABLE         = 11,
D3DRENDERSTATE_ROP2               = 12,
D3DRENDERSTATE_PLANEMASK          = 13,
D3DRENDERSTATE_ZWRITEENABLE       = 14,
D3DRENDERSTATE_ALPHATESTENABLE    = 15,
D3DRENDERSTATE_LASTPIXEL          = 16,
D3DRENDERSTATE_TEXTUREMAG         = 17,
D3DRENDERSTATE_TEXTUREMIN         = 18,
D3DRENDERSTATE_SRCBLEND           = 19,
D3DRENDERSTATE_DESTBLEND          = 20,
D3DRENDERSTATE_TEXTUREMAPBLEND    = 21,
D3DRENDERSTATE_CULLMODE           = 22,
D3DRENDERSTATE_ZFUNC              = 23,
D3DRENDERSTATE_ALPHAREF           = 24,
D3DRENDERSTATE_ALPHAFUNC          = 25,
D3DRENDERSTATE_DITHERENABLE       = 26,
D3DRENDERSTATE_ALPHABLENDENABLE   = 27,
D3DRENDERSTATE_FOGENABLE          = 28,
D3DRENDERSTATE_SPECULARENABLE     = 29,
D3DRENDERSTATE_ZVISIBLE           = 30,
D3DRENDERSTATE_SUBPIXEL           = 31,
D3DRENDERSTATE_SUBPIXELX          = 32,
D3DRENDERSTATE_STIPPLEDALPHA      = 33,
D3DRENDERSTATE_FOGCOLOR           = 34,
D3DRENDERSTATE_FOGTABLEMODE       = 35,
D3DRENDERSTATE_FOGTABLESTART      = 36,
D3DRENDERSTATE_FOGTABLEEND        = 37,
D3DRENDERSTATE_FOGTABLEDENSITY    = 38,
D3DRENDERSTATE_STIPPLEENABLE      = 39,
D3DRENDERSTATE_EDGEANTIALIAS      = 40,
D3DRENDERSTATE_COLORKEYENABLE     = 41,
D3DRENDERSTATE_BORDERCOLOR        = 43,
D3DRENDERSTATE_TEXTUREADDRESSU    = 44,
D3DRENDERSTATE_TEXTUREADDRESSV    = 45,
D3DRENDERSTATE_MIPMAPLODBIAS      = 46,
D3DRENDERSTATE_ZBIAS              = 47,
D3DRENDERSTATE_RANGEFOGENABLE     = 48,
```

```
    D3DRENDERSTATE_ANISOTROPY        = 49,
    D3DRENDERSTATE_FLUSHBATCH        = 50,  // (DX5 Only)
    D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT=51,
    D3DRENDERSTATE_STENCILENABLE     = 52,
    D3DRENDERSTATE_STENCILFAIL       = 53,
    D3DRENDERSTATE_STENCILZFAIL      = 54,
    D3DRENDERSTATE_STENCILPASS       = 55,
    D3DRENDERSTATE_STENCILFUNC       = 56,
    D3DRENDERSTATE_STENCILREF        = 57,
    D3DRENDERSTATE_STENCILMASK       = 58,
    D3DRENDERSTATE_STENCILWRITEMASK  = 59,
    D3DRENDERSTATE_TEXTUREFACTOR     = 60,
    D3DRENDERSTATE_STIPPLEPATTERN00  = 64,
      // Stipple patterns 01 through 30 omitted here.
    D3DRENDERSTATE_STIPPLEPATTERN31  = 95,
      // last line of stipple pattern
    D3DRENDERSTATE_WRAP0             = 128,
      // Wrap render states 1 through 6 omitted here.
    D3DRENDERSTATE_WRAP7             = 135,
    D3DRENDERSTATE_FORCE_DWORD       = 0x7fffffff
} D3DRENDERSTATETYPE;
```

D3DRENDERSTATE_TEXTUREHANDLE

Texture handle for use when rendering with the **Direct3DDevice2** or earlier interfaces. The default value is Nothing, which disables texture mapping and reverts to flat or Gouraud shading.

If the specified texture is in a system memory surface and the driver can only support texturing from display memory surfaces, the call will fail.

In retail builds the texture handle is not validated.

D3DRENDERSTATE_ANTIALIAS

One of the constants of the **CONST_D3DANTIALIASMODE** enumeration specifying the antialiasing of primitive edges. The default value is D3DANTIALIAS_NONE.

D3DRENDERSTATE_TEXTUREADDRESS

This render state is superseded by the D3DTSS_ADDRESS texture stage state value set through the **Direct3DDevice3.SetTextureStageState** method, but can still be used to set the addressing mode of the first texture stage. Valid values are constants of the **CONST_D3DTEXTUREADDRESS** enumeration. The default value is D3DTADDRESS_WRAP.

Applications that need to specify separate texture-addressing modes for the u and v coordinates of a texture can use the D3DRENDERSTATE_TEXTUREADDRESSU and D3DRENDERSTATE_TEXTUREADDRESSV render states.

D3DRENDERSTATE_TEXTUREPERSPECTIVE

True to enable for perspective correct texture mapping. (See perspective correction.) For the **Direct3DDevice3** interface, the default value is True.

D3DRENDERSTATE_WRAPU and
D3DRENDERSTATE_WRAPV

These render states are superseded by the D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7 render states, but can be used to set wrapping for the first texture stage. Set to True for wrapping in u direction. The default value is False.

D3DRENDERSTATE_ZENABLE

The depth buffering state, as one of the constants of the **CONST_D3DZBUFFERTYPE** enumeration. Set this state to D3DZB_TRUE to enable z-buffering, D3DZB_USEW to enable w-buffering, or D3DZB_FALSE to disable depth buffering.

The default value for this render state is D3DZB_TRUE if a depth buffer is attached to the render-target surface, and D3DZB_FALSE otherwise.

D3DRENDERSTATE_FILLMODE

One or more constants of the **CONST_D3DFILLMODE** enumeration. The default value is D3DFILL_SOLID.

D3DRENDERSTATE_SHADEMODE

One or more constants of the **D3DSHADEMODE** enumeration. The default value is D3DSHADE_GOURAUD.

D3DRENDERSTATE_MONOENABLE

True to enable monochromatic rendering, using a gray scale based on the blue channel of the color rather than full RGB. The default value is False. If the device does not support RGB rendering, the value will be True. Applications can check whether the device supports RGB rendering by using the **dcmColorModel** member of the **D3DDEVICEDESC** structure.

In monochromatic rendering, only the intensity (gray scale) component of the color and specular components are interpolated across the triangle. This means that only one channel (gray) is interpolated across the triangle instead of 3 channels (R,G,B), which is a performance gain for some hardware. This gray-scale component is derived from the blue channel of the color and specular components of the triangle.

D3DRENDERSTATE_ROP2

One of the 16 standard Windows ROP2 binary raster operations specifying how the supplied pixels are combined with the pixels of the display surface. The default value is R2_COPYPEN. Applications can use the D3DPRASTERCAPS_ROP2 flag in the **lRasterCaps** member of the **D3DPRIMCAPS** type to determine whether additional raster operations are supported.

D3DRENDERSTATE_PLANEMASK

Physical plane mask whose default value is the bitwise negation of zero (~0). This physical plane mask can be used to turn off the red bit, the blue bit, and so on. This render state is not supported by the software rasterizers, and is often ignored by hardware drivers. To disable writes to the color buffer by using alpha

blending, you can set D3DRENDERSTATE_SRCBLEND to D3DBLEND_ZERO and D3DRENDERSTATE_DESTBLEND to D3DBLEND_ONE.

D3DRENDERSTATE_ZWRITEENABLE
: True to enable writes to the depth buffer. The default value is True. This member enables an application to prevent the system from updating the depth buffer with new depth values. If this state is False, depth comparisons are still made according to the render state D3DRENDERSTATE_ZFUNC (assuming depth buffering is taking place), but depth values are not written to the buffer.

D3DRENDERSTATE_ALPHATESTENABLE
: True to enable alpha tests. The default value is False. This member enables applications to turn off the tests that otherwise would accept or reject a pixel based on its alpha value.

  The incoming alpha value is compared with the reference alpha value using the comparison function provided by the D3DRENDERSTATE_ALPHAFUNC render state. When this mode is enabled, alpha blending occurs only if the test succeeds.

D3DRENDERSTATE_LASTPIXEL
: False to enable drawing the last pixel in a line or triangle. The default value is True.

D3DRENDERSTATE_TEXTUREMAG
: This render state is superseded by the D3DTSS_MAGFILTER texture stage stage, set through the **Direct3DDevice3.SetTextureStageState** method, but can still be used to set the magnification filter for the first texture stage. This render state can be one of the constants of the **CONST_D3DTEXTUREFILTER** enumeration, which describes how a texture should be filtered when it is being magnified (that is, when a texel must cover more than one pixel). The valid values are D3DFILTER_NEAREST (the default) and D3DFILTER_LINEAR.

D3DRENDERSTATE_TEXTUREMIN
: This render state is superseded by the D3DTSS_MINFILTER texture stage stage, set through the **Direct3DDevice3.SetTextureStageState** method, but can still be used to set the minification filter for the first texture stage. This render state can be one of the constants of the **CONST_D3DTEXTUREFILTER** enumeration, which describes how a texture should be filtered when it is being made smaller (that is, when a pixel contains more than one texel). Any of the constants of the **CONST_D3DTEXTUREFILTER** enumeration can be specified for this render state. The default value is D3DFILTER_NEAREST.

D3DRENDERSTATE_SRCBLEND
: One of the constants of the **CONST_D3DBLEND** enumeration. The default value is D3DBLEND_ONE.

D3DRENDERSTATE_DESTBLEND
: One of the constants of the **CONST_D3DBLEND** enumeration. The default value is D3DBLEND_ZERO.

D3DRENDERSTATE_TEXTUREMAPBLEND

This render state is used when rendering with the **Direct3DDevice2** interface. When rendering multiple textures with the **Direct3DDevice3** interface, you can set blending operations and arguments through the **Direct3DDevice3.SetTextureStageState** method.

One of the constants of the **CONST_D3DTEXTUREBLEND** enumeration. The default value is D3DTBLEND_MODULATE.

D3DRENDERSTATE_CULLMODE

Specifies how back-facing triangles are to be culled, if at all. This can be set to one of the constants of the **CONST_D3DCULL** enumeration. The default value is D3DCULL_CCW.

D3DRENDERSTATE_ZFUNC

One of the constants of the **CONST_D3DCMPFUNC** enumeration. The default value is D3DCMP_LESSEQUAL. This member enables an application to accept or reject a pixel based on its distance from the camera.

The depth value of the pixel is compared with the depth buffer value. If the depth value of the pixel passes the comparison function, the pixel is written.

The depth value is written to the depth buffer only if the render state D3DRENDERSTATE_ZWRITEENABLE is True.

Software rasterizers and many hardware accelerators work faster if the depth test fails, since there is no need to filter and modulate the texture if the pixel is not going to be rendered.

D3DRENDERSTATE_ALPHAREF

Value specifying a reference alpha value against which pixels are tested when alpha-testing is enabled. This can be a 16:16 fixed point value (D3DFIXED) ranging from 0 to 1, inclusive, where 1.0 is represented as 0x00010000. The default value is 0.0.

D3DRENDERSTATE_ALPHAFUNC

One of the constants of the **CONST_D3DCMPFUNC** enumeration. The default value is D3DCMP_ALWAYS. This member enables an application to accept or reject a pixel based on its alpha value.

D3DRENDERSTATE_DITHERENABLE

True to enable dithering. The default value is False.

D3DRENDERSTATE_ALPHABLENDENABLE

True to enable alpha-blended transparency. The default value is False. This member supersedes the legacy D3DRENDERSTATE_BLENDENABLE render state; see remarks for more information.

You can use the D3DRENDERSTATE_COLORKEYENABLE render state to toggle color keying. (Hardware rasterizers have always used the D3DRENDERSTATE_BLENDENABLE render state only for toggling alpha blending.)

The type of alpha blending is determined by the D3DRENDERSTATE_SRCBLEND and D3DRENDERSTATE_DESTBLEND render states. D3DRENDERSTATE_ALPHABLENDENABLE, with D3DRENDERSTATE_COLORKEYENABLE, allows fine blending control.

D3DRENDERSTATE_ALPHABLENDENABLE does not affect the texture-blending modes specified by the **CONST_D3DTEXTUREBLEND** enumeration. Texture blending is logically well before the D3DRENDERSTATE_ALPHABLENDENABLE part of the pixel pipeline. The only interaction between the two is that the alpha portions remaining in the polygon after the **CONST_D3DTEXTUREBLEND** phase may be used in the D3DRENDERSTATE_ALPHABLENDENABLE phase to govern interaction with the content in the frame buffer.

Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **D3DDEVICEDESC** type to find out whether this render state is supported.

D3DRENDERSTATE_FOGENABLE
True to enable fog blending. The default value is False.

D3DRENDERSTATE_SPECULARENABLE
True to enable specular highlights. The default value is True.

Specular highlights are calculated as though every vertex in the object being lit were at the object's origin. This gives the expected results as long as the object is modeled around the origin and the distance from the light to the object is relatively large.

D3DRENDERSTATE_ZVISIBLE
(This render state is valid only for execute buffer devices.) True to enable z-checking. The default value is False. Z-checking is a culling technique in which a polygon representing the screen space of an entire group of polygons is tested against the depth-buffer to discover whether any of the polygons should be drawn.

In this mode of operation, the primitives are rendered without writing pixels or updating the depth-buffer. You can determine if the primitive would be visible by calling the **Direct3DDevice3.GetClipStatus** method. If the D3DSTATUS_ZNOTVISIBLE flag bit is set after the call, the rendered primitives are not visible, and do not need to be rendered to the frame and depth-buffers. If D3DSTATUS_ZNOTVISIBLE is not present, then one or more of the primitives are visible.

Direct3D Retained Mode uses this operation as a quick-reject test: it does the z-visible test on the bounding box of a set of primitives and only renders them if it returns True.

D3DRENDERSTATE_SUBPIXEL
True to enable subpixel correction. The default value is False.

Subpixel correction is the ability to draw pixels in precisely their correct locations. In a system that implemented subpixel correction, if a pixel were at position 0.1356, its position would be interpolated from the actual coordinate rather than simply drawn at 0 (using the integer values). Hardware can be non-subpixel correct or subpixel correct in x or in both x and y. When interpolating across the x-direction the actual coordinate is used. All hardware should be subpixel correct. Some software rasterizers are not subpixel correct because of the performance loss.

Subpixel correction means that the hardware always pre-steps the interpolant values in the x-direction to the nearest pixel centers and then steps one pixel at a time in the y-direction. For each x span it also pre-steps in the x-direction to the nearest pixel center and then steps in the x-direction one pixel each time. This results in very accurate rendering and eliminates almost all jittering of pixels on triangle edges. Most hardware either doesn't support it (always off) or always supports it (always on).

D3DRENDERSTATE_SUBPIXELX
True to enable subpixel correction in the x-direction only. The default value is False.

D3DRENDERSTATE_STIPPLEDALPHA
True to enable stippled alpha. The default value is False.

Current software rasterizers ignore this render state. You can use the D3DPSHADECAPS_ALPHAFLATSTIPPLED flag in the **D3DPRIMCAPS** type to discover whether the current hardware supports this render state.

D3DRENDERSTATE_FOGCOLOR
Value indicating RGB values, the default value is 0.

D3DRENDERSTATE_FOGTABLEMODE
The fog formula to be used for pixel fog. Set to one of the constants of the **CONST_D3DFOGMODE** enumeration. The default value is D3DFOG_NONE.

D3DRENDERSTATE_FOGTABLESTART
D3DRENDERSTATE_FOGTABLEEND
Depth at which pixel fog effects begin and end for linear fog mode. Depth can be specified in world-space or in device-space, depending on the other fog parameters. For more information, see Pixel Fog Parameters and Eye-Relative vs. Z-Based Depth.

These render states enable you to exclude fog effects for positions close to the camera. For example, you could set the starting depth to 0.3 to prevent fog effects for depths between 0.0 and 0.299, and the ending depth to 0.7 to prevent additional fog effects for depths between 0.701 and 1.0.

D3DRENDERSTATE_FOGTABLEDENSITY
Fog density for pixel fog to be used in the exponential fog modes (D3DFOG_EXP and D3DFOG_EXP2). Valid density values range from 0.0 to 1.0, inclusive. The default value is 1.0.

D3DRENDERSTATE_STIPPLEENABLE
Enables stippling in the device driver. When stippled alpha is enabled, it overrides the current stipple pattern, as specified by the D3DRENDERSTATE_STIPPLEPATTERN00 through D3DRENDERSTATE_STIPPLEPATTERN31 render states. When stippled alpha is disabled, the stipple pattern must be returned.

D3DRENDERSTATE_EDGEANTIALIAS
True to antialias lines forming the convex outline of objects. The default value is False. When set to True, only lines should be drawn. The behavior is undefined if triangles or points are drawn when this render state is set. Antialiasing is performed simply by averaging the values of neighboring pixels. Although this is

not the best way to perform antialiasing, it can be very efficient; hardware that supports this kind of operation is becoming more common.

Applications should not antialias interior edges of objects. The lines forming the outside edges should be drawn last.

D3DRENDERSTATE_COLORKEYENABLE

True to enable color-keyed transparency. The default value is False. You can use this render state with D3DRENDERSTATE_ALPHABLENDENABLE to implement fine blending control.

Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **D3DDEVICEDESC** type to find out whether this render state is supported.

When color-keyed transparency is enabled, only texture surfaces that were created with the DDSD_CKSRCBLT flag will be affected. Surfaces that were created without the DDSD_CKSRCBLT flag will exhibit color-keyed transparency effects.

D3DRENDERSTATE_BORDERCOLOR

A value specifying a border color. If the texture addressing mode is specified as D3DTADDRESS_BORDER (as set in the **CONST_D3DTEXTUREADDRESS** enumeration), this render state specifies the border color the system uses when it encounters texture coordinates outside the range [0.0, 1.0].

The format of the physical-color information specified by the value depends on the format of the DirectDraw surface.

D3DRENDERSTATE_TEXTUREADDRESSU

This render state is superseded by the D3DTSS_ADDRESSU texture stage state value set through the **Direct3DDevice3.SetTextureStageState** method, but can still be used to set the addressing mode of the first texture stage. Valid values are constants of the **CONST_D3DTEXTUREADDRESS** enumeration. The default value is D3DTADDRESS_WRAP. For more information, see Texture Addressing Modes.

This render state applies only to the u texture coordinate. This render state, along with D3DRENDERSTATE_TEXTUREADDRESSV, allows you to specify separate texture-addressing modes for the u and v coordinates of a texture. Because the D3DRENDERSTATE_TEXTUREADDRESS render state applies to both the u and v texture coordinates, it overrides any values set for the D3DRENDERSTATE_TEXTUREADDRESSU render state.

D3DRENDERSTATE_TEXTUREADDRESSV

This render state is superseded by the D3DTSS_ADDRESSV texture stage state value set through the **Direct3DDevice3.SetTextureStageState** method, but can still be used to set the addressing mode of the first texture stage. Valid values are constants of the **CONST_D3DTEXTUREADDRESS** enumeration. The default value is D3DTADDRESS_WRAP.

This render state applies only to the v texture coordinate. This render state, along with D3DRENDERSTATE_TEXTUREADDRESSU, allows you to specify separate texture-addressing modes for the u and v coordinates of a texture. Because the D3DRENDERSTATE_TEXTUREADDRESS render state applies to

both the u and v texture coordinates, it overrides any values set for the D3DRENDERSTATE_TEXTUREADDRESSV render state.

D3DRENDERSTATE_MIPMAPLODBIAS

Floating-point value used to change the level of detail (LOD) bias. This value offsets the value of the mipmap level that is computed by trilinear texturing. It is usually in the range –1.0 to 1.0; the default value is 0.0.

Each unit bias (+/-1.0) biases the selection by exactly one mipmap level. A positive bias will cause the use of larger mipmap levels, resulting in a sharper but more aliased image. A negative bias will cause the use of smaller mipmap levels, resulting in a blurrier image. Applying a negative bias also results in the referencing of a smaller amount of texture data, which can boost performance on some systems.

D3DRENDERSTATE_ZBIAS

An integer value in the range 0 to 16 that causes polygons that are physically coplanar to appear separate. Polygons with a high z-bias value will appear in front of polygons with a low value, without requiring sorting for drawing order. Polygons with a value of 1 appear in front of polygons with a value of 0, and so on. The default value is zero.

D3DRENDERSTATE_RANGEFOGENABLE

True to enable range-based vertex fog. (The default value is False, in which case the system uses depth-based fog.) In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene. In range-based fog, all fog methods work as usual, except that they use range instead of depth in the computations.

Range is the correct factor to use for fog computations, but depth is commonly used instead because range is expensive to compute and depth is generally already available. Using depth to calculate fog has the undesirable effect of having the 'fogginess' of peripheral objects change as the eye is rotated — in this case, the depth changes while the range remains constant.

This render state works only with **D3DVERTEX** vertices. When you specify **D3DLVERTEX** or **D3DTLVERTEX** vertices, the F (fog) component of the RGBF fog value should already be corrected for range.

Since no hardware currently supports per-pixel range-based fog, range correction offered only for vertex fog.

D3DRENDERSTATE_ANISOTROPY

This render state is superseded by the D3DTSS_MAXANISOTROPY texture stage state, set through the **Direct3DDevice3.SetTextureStageState** method, but can still be used to set the degree of anisotropic filtering for the first texture stage.

This render state can be an integer value that enables a degree of anisotropic filtering, used for bilinear or trilinear filtering. The value determines the maximum aspect ratio of the sampling filter kernel. To determine the range of appropriate values, use the D3DPRASTERCAPS_ANISOTROPY flag in the **D3DPRIMCAPS** structure.

Anisotropy is the distortion visible in the texels of a 3-D object whose surface is oriented at an angle with respect to the plane of the screen. The anisotropy is measured as the elongation (length divided by width) of a screen pixel that is inverse-mapped into texture space.

D3DRENDERSTATE_FLUSHBATCH
Flush any pending DrawPrimitive batches. When rendering with texture handles (using the **Direct3DDevice2** interface) you must flush batched primitives after modifying the current texture surface. Batched primitives are implicitly flushed when rendering with the **Direct3DDevice3** interface, as well as when rendering with execute buffers.

D3DRENDERSTATE_TRANSLUCENTSORTINDEPENDENT
True to enable sort-independent transparency, or False to disable.

D3DRENDERSTATE_STENCILENABLE
True to enable stenciling, or False to disable stenciling. The default value is False.

D3DRENDERSTATE_STENCILFAIL
Stencil operation to perform if the stencil test fails. This can be one of the constants of the **CONST_D3DSTENCILOP** enumeration. The default value is D3DSTENCILOP_KEEP.

D3DRENDERSTATE_STENCILZFAIL
Stencil operation to perform if the stencil test passes and depth test (z-test) fails. This can be one of the constants of the **CONST_D3DSTENCILOP** enumeration. The default value is D3DSTENCILOP_KEEP.

D3DRENDERSTATE_STENCILPASS
Stencil operation to perform if both the stencil and depth (z) tests pass. This can be one of the constants of the **CONST_D3DSTENCILOP** enumeration. The default value is D3DSTENCILOP_KEEP.

D3DRENDERSTATE_STENCILFUNC
Comparison function for the stencil test. This can be one of the constants of the **CONST_D3DCMPFUNC** enumeration. The default value is D3DCMP_ALWAYS.

The comparison function is used to compare the reference value to a stencil buffer entry. This comparison only applies to the bits in the reference value and stencil buffer entry that are set in the stencil mask (set by the D3DRENDERSTATE_STENCILMASK render state). If the comparison is true, the stencil test passes.

D3DRENDERSTATE_STENCILREF
Integer reference value for the stencil test. The default value is 0.

D3DRENDERSTATE_STENCILMASK
Mask applied to the reference value and each stencil buffer entry to determine the significant bits for the stencil test. The default mask is 0xFFFFFFFF.

D3DRENDERSTATE_STENCILWRITEMASK
Write mask applied to values written into the stencil buffer. The default mask is 0xFFFFFFFF.

D3DRENDERSTATE_TEXTUREFACTOR
Color used for multiple texture blending with the D3DTA_TFACTOR texture-blending argument or **D3DTOP_BLENDFACTORALPHA** texture-blending operation.

D3DRENDERSTATE_STIPPLEPATTERN00 through D3DRENDERSTATE_STIPPLEPATTERN31
Stipple pattern. Each render state applies to a separate line of the stipple pattern. Together, these render states specify a 32x32 stipple pattern.

D3DRENDERSTATE_WRAP0 through D3DRENDERSTATE_WRAP7
Texture wrapping behavior for multiple textures. Valid values for these render states are a combination of one or both of the D3DWRAP_U and D3DWRAP_V flags, which cause the system to wrap in the u and v directions for a given texture coordinate set.

D3DRENDERSTATE_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

Remarks

The D3DRENDERSTATE_BLENDENABLE member was superseded by the D3DRENDERSTATE_ALPHABLENDENABLE member. Its name was changed to make its meaning more explicit. To maintain compatibility with legacy applications, the D3DRENDERSTATE_BLENDENABLE constant is declared as equivalent to D3DRENDERSTATE_ALPHABLENDENABLE:

Direct3D defines the D3DRENDERSTATE_WRAPBIAS constant as a convenience for applications to enable or disable texture wrapping based on the zero-based integer of a texture coordinate set (rather than explicitly using one of the D3DRENDERSTATE_WRAP*n* state values). Add the D3DRENDERSTATE_WRAPBIAS value to the zero-based index of a texture coordinate set to calculate the D3DRENDERSTATE_WRAP*n* value that corresponds to that index, as shown in the following example:

```
// Enable U/V wrapping for textures that use the texture
// coordinate set at the index within the lIndex variable.
hr = lpD3DDevice.SetRenderState(
   lIndex + D3DRENDERSTATE_WRAPBIAS,
   D3DWRAP_U | D3DWRAPV);

// If lIndex is 3, the value that results from
// the addition equates to D3DRENDERSTATE_WRAP3 (131).
```

CONST_D3DSETSTATUSFLAGS

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_CONST_D3DSETSTATUSFLAGS_d3d_vb

The **CONST_D3DSETSTATUSFLAGS** enumeration is not used.

```
Enum CONST_D3DSETSTATUSFLAGS
   D3DSETSTATUS_ALL = 3
   D3DSETSTATUS_EXTENTS = 2
   D3DSETSTATUS_STATUS = 1
End Enum
```

D3DSETSTATUS_ALL
: Set both the status and the extents.

D3DSETSTATUS_EXTENTS
: Set the extents specified in the **drExtent** member.

D3DSETSTATUS_STATUS
: Set the status.

CONST_D3DSHADEMODE

[This is preliminary documentation and subject to change.]

The **CONST_D3DSHADEMODE** enumeration describes the supported shade mode for the D3DRENDERSTATE_SHADEMODE render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

```
Enum CONST_D3DSHADEMODE
   D3DSHADE_FLAT        = 1
   D3DSHADE_GOURAUD     = 2
   D3DSHADE_PHONG       = 3
   D3DSHADE_FORCE_DWORD  = 0x7fffffff
End Enum
```

D3DSHADE_FLAT
: Flat shade mode. The color and specular component of the first vertex in the triangle are used to determine the color and specular component of the face. These colors remain constant across the triangle; that is, they aren't interpolated.

D3DSHADE_GOURAUD
: Gouraud shade mode. The color and specular components of the face are determined by a linear interpolation between all three of the triangle's vertices.

D3DSHADE_PHONG
: Phong shade mode is not currently supported.

D3DSHADE_FORCE_DWORD
: Forces this enumeration to be 32 bits in size.

See Also

CONST_D3DRENDERSTATETYPE

IDH__dx_CONST_D3DSHADEMODE_d3d_vb

CONST_D3DSTENCILCAPSFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DSTENCILCAPSFLAGS** enumeration defines stencil buffer capability flags that are combined and present in the **lStencilCaps** member of the **D3DDEVICEDESC** type.

```
Enum CONST_D3DSTENCILCAPSFLAGS
    D3DSTENCILCAPS_DECR = 128
    D3DSTENCILCAPS_DECRSAT = 16
    D3DSTENCILCAPS_INCR = 64
    D3DSTENCILCAPS_INCRSAT = 8
    D3DSTENCILCAPS_INVERT = 32
    D3DSTENCILCAPS_KEEP = 1
    D3DSTENCILCAPS_REPLACE = 4
    D3DSTENCILCAPS_ZERO = 2
End Enum
```

D3DSTENCILCAPS_DECR
: The D3DSTENCILOP_DECR stencil buffer operation is supported.

D3DSTENCILCAPS_DECRSAT
: The D3DSTENCILOP_DECRSAT stencil buffer operation is supported.

D3DSTENCILCAPS_INCR
: The D3DSTENCILOP_INCR stencil buffer operation is supported.

D3DSTENCILCAPS_INCRSAT
: The D3DSTENCILOP_INCRSAT stencil buffer operation is supported.

D3DSTENCILCAPS_INVERT
: The D3DSTENCILOP_INVERT stencil buffer operation is supported.

D3DSTENCILCAPS_KEEP
: The D3DSTENCILOP_KEEP stencil buffer operation is supported.

D3DSTENCILCAPS_REPLACE
: The D3DSTENCILOP_REPLACE stencil buffer operation is supported.

D3DSTENCILCAPS_ZERO
: The D3DSTENCILOP_ZERO stencil buffer operation is supported.

CONST_D3DSTENCILOP

[This is preliminary documentation and subject to change.]

The **CONST_D3DSTENCILOP** enumeration describes the stencil operations for the D3DRENDERSTATE_STENCILFAIL, D3DRENDERSTATE_STENCILZFAIL, D3DRENDERSTATE_STENCILPASS render states.

IDH__dx_CONST_D3DSTENCILCAPSFLAGS_d3d_vb
IDH__dx_CONST_D3DSTENCILOP_d3d_vb

```
Enum CONST_D3DSTENCILOP
   D3DSTENCILOP_DECR = 8
   D3DSTENCILOP_DECRSAT = 5
   D3DSTENCILOP_INCR = 7
   D3DSTENCILOP_INCRSAT = 4
   D3DSTENCILOP_INVERT = 6
   D3DSTENCILOP_KEEP = 1
   D3DSTENCILOP_REPLACE = 3
   D3DSTENCILOP_ZERO = 2
End Enum
```

D3DSTENCILOP_DECR
: Decrement the stencil-buffer entry, wrapping to the maximum value if the new value is less than zero

D3DSTENCILOP_DECRSAT
: Decrement the stencil-buffer entry, clamping to zero

D3DSTENCILOP_INCRSAT
: Increment the stencil-buffer entry, clamping to the maximum value. See remarks for information on the maximum stencil-buffer values

D3DSTENCILOP_INVERT
: Invert the bits in the stencil-buffer entry.

D3DSTENCILOP_INCR
: Increment the stencil-buffer entry, wrapping to zero if the new value exceeds the maximum value. See remarks for information on the maximum stencil-buffer values

D3DSTENCILOP_KEEP
: Do not update the entry in the stencil buffer. This is the default value

D3DSTENCILOP_REPLACE
: Replace the stencil-buffer entry with reference value

D3DSTENCILOP_ZERO
: Set the stencil-buffer entry to zero

Remarks

Stencil-buffer entries are integer values ranging inclusively from 0 to 2^n -1, where *n* is the bit depth of the stencil buffer.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DTAFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DTAFLAGS** enumeration defines texture argument flags used to define texture blending stages. For details, see Texture Argument Flags.

CONST_D3DTEXOPCAPSFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DTEXOPCAPSFLAGS** enumeration defines texture-blending operation capabilities that are combined and present in the **lTextureOpCaps** member of the **D3DDEVICEDESC** type.

```
Enum CONST_D3DTEXOPCAPSFLAGS
    D3DTEXOPCAPS_ADD = 64
    D3DTEXOPCAPS_ADDSIGNED = 128
    D3DTEXOPCAPS_ADDSIGNED2X = 256
    D3DTEXOPCAPS_ADDSMOOTH = 1024
    D3DTEXOPCAPS_BLENDCURRENTALPHA = 32768
    D3DTEXOPCAPS_BLENDDIFFUSEALPHA = 2048
    D3DTEXOPCAPS_BLENDFACTORALPHA = 8192
    D3DTEXOPCAPS_BLENDTEXTUREALPHA = 4096
    D3DTEXOPCAPS_BLENDTEXTUREALPHAPM = 16384
    D3DTEXOPCAPS_BUMPENVMAP = 2097152
    D3DTEXOPCAPS_BUMPENVMAPLUMINANCE = 4194304
    D3DTEXOPCAPS_DISABLE = 1
    D3DTEXOPCAPS_DOTPRODUCT3 = 8388608
    D3DTEXOPCAPS_MODULATE = 8
    D3DTEXOPCAPS_MODULATE2X = 16
    D3DTEXOPCAPS_MODULATE4X = 32
    D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR = 131072
    D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA = 262144
    D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR = 524288
    D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA = 1048576
    D3DTEXOPCAPS_PREMODULATE = 65536
    D3DTEXOPCAPS_SELECTARG1 = 2
    D3DTEXOPCAPS_SELECTARG2 = 4
    D3DTEXOPCAPS_SUBTRACT = 512
End Enum
```

D3DTEXOPCAPS_ADD
The **D3DTOP_ADD** texture blending operation is supported by this device.

[#] IDH__dx_CONST_D3DTAFLAGS_d3d_vb
[#] IDH__dx_CONST_D3DTEXOPCAPSFLAGS_d3d_vb

D3DTEXOPCAPS_ADDSIGNED
The **D3DTOP_ADDSIGNED** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSIGNED2X
The **D3DTOP_ADDSIGNED2X** texture blending operation is supported by this device.

D3DTEXOPCAPS_ADDSMOOTH
The **D3DTOP_ADDSMOOTH** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDCURRENTALPHA
The **D3DTOP_BLENDCURRENTALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDDIFFUSEALPHA
The **D3DTOP_BLENDDIFFUSEALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDFACTORALPHA
The **D3DTOP_BLENDFACTORALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDTEXTUREALPHA
The **D3DTOP_BLENDTEXTUREALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_BLENDTEXTUREALPHAPM
The **D3DTOP_BLENDTEXTUREALPHAPM** texture blending operation is supported by this device.

D3DTEXOPCAPS_BUMPENVMAP
The **D3DTOP_BUMPENVMAP** texture blending operation is supported by this device.

D3DTEXOPCAPS_BUMPENVMAPLUMINANCE
The **D3DTOP_BUMPENVMAPLUMINANCE** texture blending operation is supported by this device.

D3DTEXOPCAPS_DISABLE
The **D3DTOP_DISABLE** texture blending operation is supported by this device.

D3DTEXOPCAPS_DOTPRODUCT3
The **D3DTOP_DOTPRODUCT3** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE
The **D3DTOP_MODULATE** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE2X
The **D3DTOP_MODULATE2X** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATE4X

The **D3DTOP_MODULATE4X** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEALPHA_ADDCOLOR
The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATECOLOR_ADDALPHA
The **D3DTOP_MODULATEALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEINVALPHA_ADDCOLOR
The **D3DTOP_MODULATEINVALPHA_ADDCOLOR** texture blending operation is supported by this device.

D3DTEXOPCAPS_MODULATEINVCOLOR_ADDALPHA
The **D3DTOP_MODULATEINVCOLOR_ADDALPHA** texture blending operation is supported by this device.

D3DTEXOPCAPS_PREMODULATE
The **D3DTOP_PREMODULATE** texture blending operation is supported by this device.

D3DTEXOPCAPS_SELECTARG1
The **D3DTOP_SELECTARG1** texture blending operation is supported by this device.

D3DTEXOPCAPS_SELECTARG2
The **D3DTOP_SELECTARG2** texture blending operation is supported by this device.

D3DTEXOPCAPS_SUBTRACT
The **D3DTOP_SUBTRACT** texture blending operation is supported by this device.

CONST_D3DTEXTUREADDRESS

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DTEXTUREADDRESS** enumeration describes the supported texture addressing modes when setting them with **Direct3DDevice3.SetTextureStageState** or with the D3DRENDERSTATE_TEXTUREADDRESS render state.

```
Enum CONST_D3DTEXTUREADDRESS
   D3DTADDRESS_WRAP        = 1
   D3DTADDRESS_MIRROR      = 2
   D3DTADDRESS_CLAMP       = 3
   D3DTADDRESS_BORDER      = 4,
   D3DTADDRESS_FORCE_DWORD  = 0x7fffffff
End Enum
```

[#] IDH__dx_CONST_D3DTEXTUREADDRESS_d3d_vb

D3DTADDRESS_WRAP
: Tile the texture at every integer junction. For example, for u values between 0 and 3, the texture will be repeated three times; no mirroring is performed.

D3DTADDRESS_MIRROR
: Similar to D3DTADDRESS_WRAP, except that the texture is flipped at every integer junction. For u values between 0 and 1, for example, the texture is addressed normally, between 1 and 2 the texture is flipped (mirrored), between 2 and 3 the texture is normal again, and so on.

D3DTADDRESS_CLAMP
: Texture coordinates outside the range [0.0, 1.0] are set to the texture color at 0.0 or 1.0, respectively.

D3DTADDRESS_BORDER
: Texture coordinates outside the range [0.0, 1.0] are set to the border color, which is a new render state corresponding to D3DRENDERSTATE_BORDERCOLOR in the **CONST_D3DRENDERSTATETYPE** enumeration.

D3DTADDRESS_FORCE_DWORD
: Forces this enumeration to be 32 bits in size.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DTEXTUREBLEND

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DTEXTUREBLEND** enumeration defines the supported texture-blending modes. This enumeration is used by the D3DRENDERSTATE_TEXTUREMAPBLEND render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

```
Enum CONST_D3DTEXTUREBLEND
   D3DTBLEND_DECAL        = 1
   D3DTBLEND_MODULATE     = 2
   D3DTBLEND_DECALALPHA   = 3
   D3DTBLEND_MODULATEALPHA = 4
   D3DTBLEND_DECALMASK    = 5
   D3DTBLEND_MODULATEMASK  = 6
   D3DTBLEND_COPY         = 7
   D3DTBLEND_ADD          = 8
   D3DTBLEND_FORCE_DWORD   = 0x7fffffff
End Enum
```

D3DTBLEND_DECAL

[#] IDH__dx_CONST_D3DTEXTUREBLEND_d3d_vb

Decal texture-blending mode is supported. In this mode, the RGB and alpha values of the texture replace the colors that would have been used with no texturing.

cPix = cTex

aPix = aTex

D3DTBLEND_MODULATE

Modulate texture-blending mode is supported. In this mode, the RGB values of the texture are multiplied with the RGB values that would have been used with no texturing. Any alpha values in the texture replace the alpha values in the colors that would have been used with no texturing; if the texture does not contain an alpha component, alpha values at the vertices in the source are interpolated between vertices.

*cPix = cSrc * cTex*

if(the texture has an alpha channel*)*

aPix = aTex

else

aPix = aSrc

D3DTBLEND_DECALALPHA

Decal-alpha texture-blending mode is supported. In this mode, the RGB and alpha values of the texture are blended with the colors that would have been used with no texturing, according to the following formulas:

*cPix = (cSrc * (1.0 - aTex)) + (aTex * cTex)*

aPix = aSrc

D3DTBLEND_MODULATEALPHA

Modulate-alpha texture-blending mode is supported. In this mode, the RGB values of the texture are multiplied with the RGB values that would have been used with no texturing, and the alpha values of the texture are multiplied with the alpha values that would have been used with no texturing.

*cPix = cSrc * cTex*

*aPix = aSrc * aTex*

D3DTBLEND_DECALMASK

This blending mode is not supported.

cPix = lsb(aTex) ? *cTex* : *cSrc*

aPix = aSrc

When the least-significant bit of the texture's alpha component is zero, the effect is as if texturing were disabled.

D3DTBLEND_MODULATEMASK

This blending mode is not supported.

cPix = lsb(aTex) ? *cTex * cSrc* : *cSrc*

aPix = aSrc

When the least-significant bit of the texture's alpha component is zero, the effect is as if texturing were disabled.

D3DTBLEND_COPY
This blending mode is obsolete, and is treated as equivalent to the D3DTBLEND_DECAL texture-blending mode.

D3DTBLEND_ADD
Add the Gouraud interpolants to the texture lookup with saturation semantics (that is, if the color value overflows it is set to the maximum possible value).

cPix = cTex + cSrc

aPix = aSrc

D3DTBLEND_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

Remarks

In the formulas given for the constants of this enumeration, the placeholders have the following meanings:

- *cTex* is the color of the source texel
- *aTex* is the alpha component of the source texel
- *cSrc* is the interpolated color of the source primitive
- *aSrc* is the alpha component of the source primitive
- *cPix* is the new blended color value
- *aPix* is the new blended alpha value

Modulation combines the effects of lighting and texturing. Because colors are specified as values between and including 0 and 1, modulating (multiplying) the texture and preexisting colors together typically produces colors that are less bright than either source. The brightness of a color component is undiminished when one of the sources for that component is white (1). The simplest way to ensure that the colors of a texture do not change when the texture is applied to an object is to ensure that the object is white (1,1,1).

CONST_D3DTEXTUREFILTER

[This is preliminary documentation and subject to change.]

The **CONST_D3DTEXTUREFILTER** enumeration defines the supported texture filter modes used by the D3DRENDERSTATE_TEXTUREMAG render state in the **CONST_D3DRENDERSTATETYPE** enumeration.

Enum CONST_D3DTEXTUREFILTER

IDH__dx_CONST_D3DTEXTUREFILTER_d3d_vb

```
    D3DFILTER_NEAREST        = 1
    D3DFILTER_LINEAR         = 2
    D3DFILTER_MIPNEAREST     = 3
    D3DFILTER_MIPLINEAR      = 4
    D3DFILTER_LINEARMIPNEAREST = 5
    D3DFILTER_LINEARMIPLINEAR  = 6
    D3DFILTER_FORCE_DWORD    = 0x7fffffff
End Enum
```

D3DFILTER_NEAREST
: The texel with coordinates nearest to the desired pixel value is used. This is a point filter with no mipmapping.

  This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DFILTER_LINEAR
: A weighted average of a 2×2 area of texels surrounding the desired pixel is used. This is a bilinear filter with no mipmapping.

  This applies to both zooming in and zooming out. If either zooming in or zooming out is supported, then both must be supported.

D3DFILTER_MIPNEAREST
: The closest mipmap level is chosen and a point filter is applied.

D3DFILTER_MIPLINEAR
: The closest mipmap level is chosen and a bilinear filter is applied within it.

D3DFILTER_LINEARMIPNEAREST
: The two closest mipmap levels are chosen and then a linear blend is used between point filtered samples of each level.

D3DFILTER_LINEARMIPLINEAR
: The two closest mipmap levels are chosen and then combined using a bilinear filter.

D3DFILTER_FORCE_DWORD
: Forces this enumeration to be 32 bits in size.

Remarks

All of these filter modes are valid with the D3DRENDERSTATE_TEXTUREMIN render state, but only the first two (D3DFILTER_NEAREST and D3DFILTER_LINEAR) are valid with D3DRENDERSTATE_TEXTUREMAG.

CONST_D3DTEXTUREMAGFILTER

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_CONST_D3DTEXTUREMAGFILTER_d3d_vb

The **CONST_D3DTEXTUREMAGFILTER** enumeration defines texture magnification filtering modes for a texture stage.

```
Enum CONST_D3DTEXTUREMAGFILTER
    D3DTFG_POINT        = 1,
    D3DTFG_LINEAR       = 2,
    D3DTFG_FLATCUBIC    = 3,
    D3DTFG_GAUSSIANCUBIC= 4,
    D3DTFG_ANISOTROPIC  = 5,
    D3DTFG_FORCE_DWORD  = 0x7fffffff,
End Enum
```

D3DTFG_POINT
: Point filtering. The texel with coordinates nearest to the desired pixel value is used.

D3DTFG_LINEAR
: Bilinear interpolation filtering. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

D3DTFG_FLATCUBIC
: Not currently supported; do not use.

D3DTFG_GAUSSIANCUBIC
: Not currently supported; do not use.

D3DTFG_ANISOTROPIC
: Anisotropic texture filtering. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

D3DTFG_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size.

Remarks

You set a texture stage's magnification filter by calling the **Direct3DDevice3.SetTextureStageState** method with the D3DTSS_MAGFILTER value as the second parameter, and one of constants of this enumeration as the third parameter.

See Also

CONST_D3DTEXTUREMINFILTER, **CONST_D3DTEXTUREMIPFILTER**

CONST_D3DTEXTUREMINFILTER

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_CONST_D3DTEXTUREMINFILTER_d3d_vb

The **CONST_D3DTEXTUREMINFILTER** enumeration defines texture minification filtering modes for a texture stage.

```
Enum CONST_D3DTEXTUREMINFILTER
   D3DTFN_POINT        = 1,
   D3DTFN_LINEAR       = 2,
   D3DTFN_ANISOTROPIC  = 3,
   D3DTFN_FORCE_DWORD  = 0x7fffffff,
End Enum
```

D3DTFN_POINT
: Point filtering. The texel with coordinates nearest to the desired pixel value is used.

D3DTFN_LINEAR
: Bilinear interpolation filtering. A weighted average of a 2×2 area of texels surrounding the desired pixel is used.

D3DTFN_ANISOTROPIC
: Anisotropic texture filtering. Compensates for distortion caused by the difference in angle between the texture polygon and the plane of the screen.

D3DTFN_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size.

Remarks

You set a texture stage's magnification filter by calling the **Direct3DDevice3.SetTextureStageState** method with the D3DTSS_MINFILTER value as the second parameter, and one of constants of this enumeration as the third parameter.

See Also

CONST_D3DTEXTUREMAGFILTER, **CONST_D3DTEXTUREMIPFILTER**

CONST_D3DTEXTUREMIPFILTER

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DTEXTUREMIPFILTER** enumeration defines texture mipmap filtering modes for a texture stage.

```
Enum CONST_D3DTEXTUREMIPFILTER
   D3DTFP_NONE         = 1,
   D3DTFP_POINT        = 2,
   D3DTFP_LINEAR       = 3,
   D3DTFP_FORCE_DWORD  = 0x7fffffff,
```

IDH__dx_CONST_D3DTEXTUREMIPFILTER_d3d_vb

```
End Enum
```

D3DTFP_NONE
: Mipmapping disabled. The rasterizer should use the magnification filter instead.

D3DTFP_POINT
: Nearest point mipmap filtering. The rasterizer uses the color from the texel of the nearest mipmap texture.

D3DTFP_LINEAR
: Trilinear mipmap interpolation. The rasterizer linearly interpolates pixel color using the texels of the two nearest mipmap textures.

D3DTFP_FORCE_DWORD
: Forces this enumeration to compile to 32 bits in size.

Remarks

You set a texture stage's magnification filter by calling the **Direct3DDevice3.SetTextureStageState** method with the D3DTSS_MIPFILTER value as the second parameter, and one of constants of this enumeration as the third parameter.

See Also

CONST_D3DTEXTUREMAGFILTER, **CONST_D3DTEXTUREMIPFILTER**

CONST_D3DTEXTUREOP

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DTEXTUREOP** enumeration defines per-stage texture blending operations. The constants of this type are used when setting color or alpha operations by using the D3DTSS_COLOROP or D3DTSS_ALPHAOP values with the **Direct3DDevice3.SetTextureStageState** method.

```
Enum CONST_D3DTEXTUREOP
    D3DTOP_DISABLE    = 1,
    D3DTOP_SELECTARG1 = 2,
    D3DTOP_SELECTARG2 = 3,
    D3DTOP_MODULATE   = 4,
    D3DTOP_MODULATE2X = 5,
    D3DTOP_MODULATE4X = 6,
    D3DTOP_ADD        = 7,
    D3DTOP_ADDSIGNED  = 8,
    D3DTOP_ADDSIGNED2X  =  9,
    D3DTOP_SUBTRACT     = 10,
    D3DTOP_ADDSMOOTH    = 11,
```

[#] IDH__dx_CONST_D3DTEXTUREOP_d3d_vb

```
    D3DTOP_BLENDDIFFUSEALPHA    = 12,
    D3DTOP_BLENDTEXTUREALPHA    = 13,
    D3DTOP_BLENDFACTORALPHA     = 14,
    D3DTOP_BLENDTEXTUREALPHAPM  = 15,
    D3DTOP_BLENDCURRENTALPHA    = 16,
    D3DTOP_PREMODULATE          = 17,
    D3DTOP_MODULATEALPHA_ADDCOLOR = 18,
    D3DTOP_MODULATECOLOR_ADDALPHA = 19,
    D3DTOP_MODULATEINVALPHA_ADDCOLOR = 20,
    D3DTOP_MODULATEINVCOLOR_ADDALPHA = 21,
    D3DTOP_BUMPENVMAP           = 22
    D3DTOP_BUMPENVMAPLUMINANCE  = 23
    D3DTOP_DOTPRODUCT3          = 24,
    D3DTOP_FORCE_DWORD = 0x7fffffff,
End Enum
```

Control constants

D3DTOP_DISABLE

Disables output from this texture stage and all stages with a higher index.

D3DTOP_SELECTARG1

Use this texture stage's first color or alpha argument, unmodified, as the output. This operation affects the color argument when used with the D3DTSS_COLOROP texture stage state, and the alpha argument when used with D3DTSS_ALPHAOP.

$S_{RGBA} = Arg1$

D3DTOP_SELECTARG2

Use this texture stage's second color or alpha argument, unmodified, as the output. This operation affects the color argument when used with the D3DTSS_COLOROP texture stage state, and the alpha argument when used with D3DTSS_ALPHAOP.

$S_{RGBA} = Arg2$

Modulation constants

D3DTOP_MODULATE

Multiply the components of the arguments together.

$S_{RGBA} = Arg1 \times Arg2$

D3DTOP_MODULATE2X

Multiply the components of the arguments and shift the products to the left one bit (effectively multiplying them by two) for brightening.

$S_{RGBA} = (Arg1 \times Arg2) << 1$

D3DTOP_MODULATE4X

Multiply the components of the arguments and shift the products to the left two bits (effectively multiplying them by four) for brightening.

$S_{RGBA} = (Arg1 \times Arg2) << 2$

Addition and Subtraction constants

D3DTOP_ADD

Add the components of the arguments.

$S_{RGBA} = Arg1 + Arg2$

D3DTOP_ADDSIGNED

Add the components of the arguments with a -0.5 bias, making the effective range of values from -0.5 to 0.5.

$S_{RGBA} = Arg1 + Arg2 - 0.5$

D3DTOP_ADDSIGNED2X

Add the components of the arguments with a -0.5 bias, and shift the products to the left one bit.

$S_{RGBA} = (Arg1 + Arg2 - 0.5) << 2$

D3DTOP_SUBTRACT

Subtract the components of the second argument from those of the first argument.

$S_{RGBA} = Arg1 - Arg2$

D3DTOP_ADDSMOOTH

Add the first and second arguments, then subract their product from the sum.

$S_{RGBA} = Arg\ 1 + Arg2 - Arg1 \times Arg2$
$= Arg1 + Arg2\ (1 - Arg1)$

Linear alpha blending constants

D3DTOP_BLENDDIFFUSEALPHA
D3DTOP_BLENDTEXTUREALPHA
D3DTOP_BLENDFACTORALPHA
D3DTOP_BLENDCURRENTALPHA

Linearly blend this texture stage using the interpolated alpha from each vertex (D3DTOP_BLENDDIFFUSEALPHA), alpha from this stage's texture (D3DTOP_BLENDTEXTUREALPHA), a scalar alpha (D3DTOP_BLENDFACTORALPHA) set with the D3DRENDERSTATE_TEXTUREFACTOR render state, or the alpha taken from the previous texture stage (D3DTOP_BLENDCURRENTALPHA).

$S_{RGBA} = Arg\ 1 \times (Alpha) + Arg2 \times (1 - Alpha)$

D3DTOP_BLENDTEXTUREALPHAPM

Linearly blend a texture stage that uses premultiplied alpha.

$S_{RGBA} = Arg\ 1 + Arg2 \times (1 - Alpha)$

Specular mapping constants

D3DTOP_PREMODULATE
Modulate this texture stage with the next texture stage.

D3DTOP_MODULATEALPHA_ADDCOLOR
Modulate the second argument's color using the first argument's alpha, then add the result to argument one. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = Arg\ 1_{RGB} + Arg1_{A} \times Arg2_{RGB}$

D3DTOP_MODULATECOLOR_ADDALPHA
Modulate the arguments, then add the first argument's alpha. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = Arg\ 1_{RGB} \times Arg2_{RGB} + Arg1_{A}$

D3DTOP_MODULATEINVALPHA_ADDCOLOR
Similar to D3DTOP_MODULATEALPHA_ADDCOLOR, but use the inverse of the first argument's alpha. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = (1 - Arg\ 1_{A}) \times Arg2_{RGB} + Arg1_{RGB}$

D3DTOP_MODULATEINVCOLOR_ADDALPHA
Similar to D3DTOP_MODULATECOLOR_ADDALPHA, but use the inverse of the first argument's color. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = (1 - Arg\ 1_{RGB}) \times Arg2_{RGB} + Arg1_{A}$

Bump mapping constants

D3DTOP_BUMPENVMAP
Perform per-pixel bump-mapping using the environment map in the next texture stage (without luminance).

D3DTOP_BUMPENVMAPLUMINANCE
Perform per-pixel bump-mapping using the environment map in the next texture stage (with luminance).

D3DTOP_DOTPRODUCT3
Modulate the components of each argument (as signed components), add their products, then replicate the sum to all color channels, including alpha. This operation is supported only for color operations (D3DTSS_COLOROP).

$S_{RGBA} = (Arg\ 1_{R} \times Arg2_{R} + Arg1_{G} \times Arg2_{G} + Arg1_{B} \times Arg2_{B})$

Miscellaneous member

D3DTOP_FORCE_DWORD
Forces this enumeration to be compiled to 32 bits in size. This value is not used.

Remarks

In the preceding formulas, S_{RGBA} is the RGBA color produced by a texture operation, and *Arg1* and *Arg2* represent the complete RGBA color of the texture arguments. Individual components of an argument are shown with subscripts. For example, the alpha component for argument one would be shown as $Arg1_A$.

See Also

Direct3DDevice3.GetTextureStageState, **Direct3DDevice3.SetTextureStageState**, **CONST_D3DTEXTURESTAGESTATETYPE**

CONST_D3DTEXTURESTAGESTATETYPE

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DTEXTURESTAGESTATETYPE** enumeration defines texture stage states. Constants of this enumeration are used with the **Direct3DDevice3.GetTextureStageState** and **Direct3DDevice3.SetTextureStageState** methods to retrieve and set texture state values.

```
Enum CONST_D3DTEXTURESTAGESTATETYPE
    D3DTSS_COLOROP        = 1,
    D3DTSS_COLORARG1      = 2,
    D3DTSS_COLORARG2      = 3,
    D3DTSS_ALPHAOP        = 4,
    D3DTSS_ALPHAARG1      = 5,
    D3DTSS_ALPHAARG2      = 6,
    D3DTSS_BUMPENVMAT00   = 7,
    D3DTSS_BUMPENVMAT01   = 8,
    D3DTSS_BUMPENVMAT10   = 9,
    D3DTSS_BUMPENVMAT11   = 10
    D3DTSS_TEXCOORDINDEX  = 11
    D3DTSS_ADDRESS        = 12
    D3DTSS_ADDRESSU       = 13
    D3DTSS_ADDRESSV       = 14
    D3DTSS_BORDERCOLOR    = 15
    D3DTSS_MAGFILTER      = 16
    D3DTSS_MINFILTER      = 17
```

[#] IDH__dx_CONST_D3DTEXTURESTAGESTATETYPE_d3d_vb

```
    D3DTSS_MIPFILTER      = 18
    D3DTSS_MIPMAPLODBIAS  = 19
    D3DTSS_MAXMIPLEVEL    = 20
    D3DTSS_MAXANISOTROPY  = 21
    D3DTSS_BUMPENVLSCALE  = 22
    D3DTSS_BUMPENVLOFFSET = 23
    D3DTSS_FORCE_DWORD    = 0x7fffffff,
End Enum
```

D3DTSS_COLOROP
: The texture stage state is a texture color blending operation identified by one of the constants of the **CONST_D3DTEXTUREOP** enumeration. The default value for the first texture stage (stage zero) is D3DTOP_MODULATE, and for all other stages the default is D3DTOP_DISABLE.

D3DTSS_COLORARG1
: The texture stage state is the first color argument for the stage, identified by a texture argument flag. The default argument is D3DTA_TEXTURE.

D3DTSS_COLORARG2
: The texture stage state is the second color argument for the stage, identified by a texture argument flag. The default argument is D3DTA_CURRENT.

D3DTSS_ALPHAOP
: The texture stage state is texture alpha blending operation identified by one of the constants of the **CONST_D3DTEXTUREOP** enumeration. The default value for the first texture stage (stage zero) is D3DTOP_SELECTARG1, and for all other stages the default is D3DTOP_DISABLE.

D3DTSS_ALPHAARG1
: The texture stage state is the first alpha argument for the stage, identified by a texture argument flag. The default argument is D3DTA_TEXTURE. If no texture is set for this stage, the default argument is D3DTA_DIFFUSE.

D3DTSS_ALPHAARG2
: The texture stage state is the second alpha argument for the stage, identified by a texture argument flag. The default argument is D3DTA_CURRENT.

D3DTSS_BUMPENVMAT00
: The texture stage state is a value for the [0][0] coefficient in a bump mapping matrix. The default value is zero.

D3DTSS_BUMPENVMAT01
: The texture stage state is a value for the [0][1] coefficient in a bump mapping matrix. The default value is 0.

D3DTSS_BUMPENVMAT10
: The texture stage state is a value for the [1][0] coefficient in a bump mapping matrix. The default value is 0.

D3DTSS_BUMPENVMAT11
: The texture stage state is a value for the [1][1] coefficient in a bump mapping matrix. The default value is 0.

D3DTSS_TEXCOORDINDEX

Index of the texture coordinate set to use with this texture stage. The default index is 0. Set this state to the zero-based index of the texture set at for each vertex that this texture stage will use. (You can specify up to eight sets of texture coordinates per vertex.) If a vertex does not include a set of texture coordinates at the specified index, the system defaults to using the u, v coordinates (0,0).

D3DTSS_ADDRESS
Member of the **CONST_D3DTEXTUREADDRESS** enumeration. Selects the texture addressing method for both the u and v coordinates. The default is D3DTADDRESS_WRAP.

D3DTSS_ADDRESSU
Member of the **CONST_D3DTEXTUREADDRESS** enumeration. Selects the texture addressing method for the u coordinate. The default is D3DTADDRESS_WRAP.

D3DTSS_ADDRESSV
Member of the **CONST_D3DTEXTUREADDRESS** enumeration. Selects the texture addressing method for the v coordinate. The default value is D3DTADDRESS_WRAP.

D3DTSS_BORDERCOLOR
Value that describes the color to be used for rasterizing texture coordinates outside the [0.0,1.0] range. The default color is 0x00000000.

D3DTSS_MAGFILTER
Member of the **CONST_D3DTEXTUREMAGFILTER** enumeration that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFG_POINT.

D3DTSS_MINFILTER
Member of the **CONST_D3DTEXTUREMINFILTER** enumeration that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFN_POINT.

D3DTSS_MIPFILTER
Member of the **CONST_D3DTEXTUREMIPFILTER** enumeration that indicates the texture magnification filter to be used when rendering the texture onto primitives. The default value is D3DTFP_NONE.

D3DTSS_MIPMAPLODBIAS
Level of detail bias for mipmaps. Can be used to make textures appear more chunky or more blurred. The default value is 0.

D3DTSS_MAXMIPLEVEL
Maximum level-of-detail mipmap that the application will allow. Zero, which is the default, indicates that all levels can be used.

D3DTSS_MAXANISOTROPY
Maximum level of anisotropy. The default value is 1.

D3DTSS_BUMPENVLSCALE
Scale for bump map luminance. The default value is 0.

D3DTSS_BUMPENVLOFFSET
Offset for bump map luminance. The default value is 0.

D3DTSS_FORCE_DWORD
Forces this enumeration to be compiled to 32 bits in size. This value is not used.

Remarks

The valid range of values for the D3DTSS_BUMPENVMAT00, D3DTSS_BUMPENVMAT01, D3DTSS_BUMPENVMAT10, and D3DTSS_BUMPENVMAT11 bump-mapping matrix coefficients is greater than or equal to -8.0, and less than 8.0. This range, expressed in mathematical notation is [-8.0,8.0).

CONST_D3DTRANSFORMCAPS

[This is preliminary documentation and subject to change.]

The **CONST_D3DTRANSFORMCAPS** enumeration describes the transformation capabilities of the device. This enumeration is used in the **lTransformCaps** member of the **D3DDEVICEDESC** type.

```
Enum CONST_D3DTRANSFORMCAPS
   D3DTRANSFORMCAPS_CLIP = 1
End Enum
```

D3DTRANSFORMCAPS_CLIP
The system clips while transforming.

CONST_D3DTRANSFORMFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DTRANSFORMFLAGS** enumeration defines that are used in the **flags** parameter of the **Direct3DViewport3.TransformVertices** method.

```
Enum CONST_D3DTRANSFORMFLAGS
   D3DTRANSFORM_CLIPPED = 1
   D3DTRANSFORM_UNCLIPPED = 2
End Enum
```

D3DTRANSFORM_CLIPPED
Transform the vertices and adjust the rectangle in the **rExtent** member of the associated **D3DTRANSFORMDATA** type to reflect the new extents.

D3DTRANSFORM_UNCLIPPED
Transform the vertices without updating the extents.

IDH__dx_CONST_D3DTRANSFORMCAPS_d3d_vb
IDH__dx_CONST_D3DTRANSFORMFLAGS_d3d_vb

CONST_D3DTRANSFORMSTATETYPE

[This is preliminary documentation and subject to change.]

The **CONST_D3DTRANSFORMSTATETYPE** enumeration defines values that describe the transformation state.

```
Enum CONST_D3DTRANSFORMSTATETYPE
   D3DTRANSFORMSTATE_WORLD       = 1
   D3DTRANSFORMSTATE_VIEW        = 2
   D3DTRANSFORMSTATE_PROJECTION   = 3
   D3DTRANSFORMSTATE_FORCE_DWORD  = 0x7fffffff
End Enum
```

D3DTRANSFORMSTATE_WORLD, D3DTRANSFORMSTATE_VIEW, and D3DTRANSFORMSTATE_PROJECTION
: Define the matrices for the world, view, and projection transformations. The default values are Nothing (the identity matrices).

D3DTRANSFORMSTATE_FORCE_DWORD
: Forces this enumeration to be 32 bits in size.

See Also

CONST_D3DRENDERSTATETYPE

CONST_D3DVBCAPSFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DVBCAPSFLAGS** enumeration defines vertex buffer capability flags that are used in the lCaps member of the **D3DVERTEXBUFFERDESC** type.

```
Enum CONST_D3DVBCAPSFLAGS
   D3DVBCAPS_OPTIMIZED = 2147483648
   D3DVBCAPS_SYSTEMMEMORY = 2048
   D3DVBCAPS_WRITEONLY = 65536
End Enum
```

D3DVBCAPS_OPTIMIZED
: The vertex buffer contains optimized vertex data. (This flag is not used when creating a new vertex buffer.)

D3DVBCAPS_SYSTEMMEMORY

IDH__dx_CONST_D3DTRANSFORMSTATETYPE_d3d_vb
IDH__dx_CONST_D3DVBCAPSFLAGS_d3d_vb

The vertex buffer should be created in system memory. Use this capability for vertex buffers that will be rendered by using software devices (MMX and RGB devices).

D3DVBCAPS_WRITEONLY
Hints to the system that the application will only write to the vertex buffer. Using this flag enables the driver to choose the best memory location for efficient write operations and rendering. Attempts to read from a vertex buffer that is created with this capability can result in degraded performance.

CONST_D3DVERTEXTYPE

[This is preliminary documentation and subject to change.]

The **CONST_D3DVERTEXTYPE** enumeration lists the vertex types that are supported by the legacy **Direct3DDevice2** and **Direct3DDevice** interfaces. If your application uses **Direct3DDevice3**, the **CONST_D3DVERTEXTYPE** enumeration is superseded by flexible vertex format flags.

```
Enum CONST_D3DVERTEXTYPE
   D3DVT_VERTEX      = 1
   D3DVT_LVERTEX     = 2
   D3DVT_TLVERTEX    = 3
   D3DVT_FORCE_DWORD  = 0x7fffffff
End Enum
```

D3DVT_VERTEX
All the vertices in the array are of the **D3DVERTEX** type. This setting will cause transformation, lighting and clipping to be applied to the primitive as it is rendered.

D3DVT_LVERTEX
All the vertices in the array are of the **D3DLVERTEX** type. When used with this option, the primitive will have transformations applied during rendering.

D3DVT_TLVERTEX
All the vertices in the array are of the **D3DTLVERTEX** type. Rasterization only will be applied to this data.

D3DVT_FORCE_DWORD
Forces this enumeration to be 32 bits in size.

CONST_D3DVISFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_D3DVISFLAGS** enumeration defines flags used with the **Direct3DDevice3.ComputeSphereVisibility** method.

IDH__dx_CONST_D3DVERTEXTYPE_d3d_vb
IDH__dx_CONST_D3DVISFLAGS_d3d_vb

```
Enum CONST_D3DVISFLAGS
   D3DVIS_INSIDE_BOTTOM = 0
   D3DVIS_INSIDE_FAR = 0
   D3DVIS_INSIDE_FRUSTUM = 0
   D3DVIS_INSIDE_LEFT = 0
   D3DVIS_INSIDE_NEAR = 0
   D3DVIS_INSIDE_RIGHT = 0
   D3DVIS_INSIDE_TOP = 0
   D3DVIS_INTERSECT_BOTTOM = 256
   D3DVIS_INTERSECT_FAR = 4096
   D3DVIS_INTERSECT_FRUSTUM = 1
   D3DVIS_INTERSECT_LEFT = 2
   D3DVIS_INTERSECT_NEAR = 1024
   D3DVIS_INTERSECT_RIGHT = 16
   D3DVIS_INTERSECT_TOP = 64
   D3DVIS_MASK_BOTTOM = 768
   D3DVIS_MASK_FAR = 12288
   D3DVIS_MASK_FRUSTUM = 3
   D3DVIS_MASK_LEFT = 12
   D3DVIS_MASK_NEAR = 3072
   D3DVIS_MASK_RIGHT = 40
   D3DVIS_MASK_TOP = 192
   D3DVIS_OUTSIDE_BOTTOM = 512
   D3DVIS_OUTSIDE_FAR = 8192
   D3DVIS_OUTSIDE_FRUSTUM = 2
   D3DVIS_OUTSIDE_LEFT = 4
   D3DVIS_OUTSIDE_NEAR = 2048
   D3DVIS_OUTSIDE_RIGHT = 32
   D3DVIS_OUTSIDE_TOP = 128
End Enum
```

Inside flags

D3DVIS_INSIDE_BOTTOM, D3DVIS_INSIDE_FAR,
D3DVIS_INSIDE_FRUSTUM, D3DVIS_INSIDE_LEFT,
D3DVIS_INSIDE_NEAR, D3DVIS_INSIDE_RIGHT, D3DVIS_INSIDE_TOP
: The sphere is inside the viewing frustum of the current viewport.

Intersection flags

D3DVIS_INTERSECT_BOTTOM or D3DVIS_INTERSECT_TOP
: The sphere intersects the bottom or top plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_INTERSECT_FAR or D3DVIS_INTERSECT_NEAR
: The sphere intersects the far or near plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_INTERSECT_FRUSTUM

The sphere intersects some part of the viewing frustum for the current viewport.

D3DVIS_INTERSECT_LEFT or D3DVIS_INTERSECT_RIGHT
The sphere intersects the left or right plane of the viewing frustum for the current viewport, depending on which flag is present.

Outside flags

D3DVIS_OUTSIDE_BOTTOM or D3DVIS_OUTSIDE_TOP
The sphere is outside the bottom or top plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_OUTSIDE_FAR or D3DVIS_OUTSIDE_NEAR
The sphere is outside the far or near plane of the viewing frustum for the current viewport, depending on which flag is present.

D3DVIS_OUTSIDE_FRUSTUM
The sphere is somewhere outside the viewing frustum for the current viewport.

D3DVIS_OUTSIDE_LEFT or D3DVIS_OUTSIDE_RIGHT
The sphere is outside the left or right plane of the viewing frustum for the current viewport, depending on which flag is present.

CONST_D3DVOPFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_D3DVOPFLAGS** enumeration devices vertex operation flags used in the *vertexOp* parameter of the **Direct3DVertexBuffer.ProcessVertices** method.

```
Enum CONST_D3DVOPFLAGS
   D3DVOP_CLIP      = =2
   D3DVOP_EXTENTS   = =4
   D3DVOP_LIGHT     = 1024
   D3DVOP_TRANSFORM = =1
End Enum
```

D3DVOP_CLIP
Transform the vertices and clip any vertices that exist outside the viewing frustum. This flag cannot be used with vertex buffers that do not contain clipping information (for example, created with the D3DDP_DONOTCLIP flag).

D3DVOP_EXTENTS
Transform the vertices, then update the extents of the screen rectangle when the vertices are rendered. Using this flag can potentially help performance, but the extents returned by **Direct3DDevice3.GetClipStatus** will not have been updated to account for the vertices when they are rendered.

D3DVOP_LIGHT
Light the vertices.

D3DVOP_TRANSFORM

[#] IDH__dx_CONST_D3DVOPFLAGS_d3d_vb

Transform the vertices using the world, view, and projection matrices. This flag must always be set.

CONST_D3DZBUFFERTYPE

[This is preliminary documentation and subject to change.]

The **CONST_D3DZBUFFERTYPE** enumeration describes depth-buffer formats for use with the D3DRENDERSTATE_ZENABLE render state.

```
Enum CONST_D3DZBUFFERTYPE
   D3DZB_FALSE       = 0,
   D3DZB_TRUE        = 1,
   D3DZB_USEW        = 2,
   D3DZB_FORCE_DWORD  = 0x7fffffff
End Enum
```

D3DZB_FALSE
Disable depth-buffering.

D3DZB_TRUE
Enable z-buffering.

D3DZB_USEW
Enable w-buffering.

D3DZB_FORCE_DWORD
Forces this enumeration to be compiled to 32-bits in size. This value is not used.

Remarks

The D3DZB_FALSE and D3DZB_TRUE values are interchangeable with the True and False macro values previously used with D3DRENDERSTATE_ZENABLE.

See Also

Direct3DDevice3.SetRenderState, Depth Buffers

Flexible Vertex Format Flags

[This is preliminary documentation and subject to change.]

Direct3D Immediate Mode uses flag values to describe vertex formats used for DrawPrimitive-based rendering. The **CONST_D3DFVFFLAGS** enumeration defines the following flags to explicitly describe a vertex format, and provides helper macros that act as common combinations of such flags. For more information, see About Vertex Formats.

Flexible vertex format (FVF) flags

IDH__dx_CONST_D3DZBUFFERTYPE_d3d_vb

D3DFVF_DIFFUSE
Vertex format includes a diffuse color component.

D3DFVF_NORMAL
Vertex format includes a vertex normal vector. This flag cannot be used with the D3DFVF_XYZRHW flag.

D3DFVF_SPECULAR
Vertex format includes a specular color component.

D3DFVF_XYZ
Vertex format includes the position of an untransformed vertex. This flag cannot be used with the D3DFVF_XYZRHW flag. If you use this flag, you must also specify a vertex normal, a vertex color component (D3DFVF_DIFFUSE or D3DFVF_SPECULAR), or include at least one set of texture coordinates (D3DFVF_TEX1 through D3DFVF_TEX8).

D3DFVF_XYZRHW
Vertex format includes the position of a transformed vertex. This flag cannot be used with the D3DFVF_XYZ or D3DFVF_NORMAL flags. If you use this flag, you must also specify a vertex color component (D3DFVF_DIFFUSE or D3DFVF_SPECULAR) or include at least one set of texture coordinates (D3DFVF_TEX1 through D3DFVF_TEX8).

Texture-related FVF flags

D3DFVF_TEX0 through D3DFVF_TEX8
Number of texture coordinate sets for this vertex. The actual values for these flags are not sequential.

Helper macros

D3DFVF_LVERTEX
Vertex format is equivalent to the **D3DLVERTEX** vertex type.

D3DFVF_TLVERTEX
Vertex format is equivalent to the **D3DTLVERTEX** vertex type.

D3DFVF_VERTEX
Vertex format is equivalent to the **D3DVERTEX** vertex type.

Mask values

D3DFVF_POSITION_MASK
Mask for position bits.

D3DFVF_RESERVED0 and D3DFVF_RESERVED2
Mask values for reserved bits in the flexible vertex format.

D3DFVF_RESERVED1,
This bit is reserved to indicate that the system should emulate D3DLVERTEX processing. If this flag is used, the D3DFVF_XYZ, D3DFVF_DIFFUSE, D3DFVF_SPECULAR, and D3DFVF_TEX1 flags must also be used. This equates to the effect of the **D3DFVF_LVERTEX** helper macro.

D3DFVF_TEXCOUNT_MASK
Mask value for texture flag bits.

Miscellaneous

D3DFVF_TEXCOUNT_SHIFT

The number of bits to shift an integer value that identifies the number of a texture coordinates for a vertex.

Texture Argument Flags

[This is preliminary documentation and subject to change.]

Each texture stage for a device can have two texture arguments that affect the color or alpha channel of the texture. You set and retrieve texture arguments by calling the **Direct3DDevice3.SetTextureStageState** and **Direct3DDevice3.GetTextureStageState**, specifying the D3DTSS_COLORARG1, D3DTSS_COLORARG2, D3DTSS_ALPHAARG1 or D3DTSS_ALPHAARG2 constants of the **CONST_D3DTEXTURESTAGESTATETYPE** enumeration.

The following flags, organized as arguments and modifiers, can be used with color and alpha arguments for a texture stage. You can combine an argument flag with a modifier, but you cannot combine two argument flags.

Argument flags

D3DTA_CURRENT
: The texture argument is the result of the previous blending stage. In the first texture stage (stage zero), this argument defaults to D3DTA_DIFFUSE.

D3DTA_DIFFUSE
: The texture argument is the diffuse color interpolated from vertex components during Gouraud shading. If the vertex does not contain a diffuse color, the default color is 0xFFFFFFFF.

D3DTA_SELECTMASK
: Mask value for all arguments; not used when setting texture arguments.

D3DTA_TEXTURE
: The texture argument is the texture color for this texture stage. This is valid only for the first color and alpha arguments in a stage (the D3DTSS_COLORARG1 and D3DTSS_ALPHAARG1 constants of **CONST_D3DTEXTURESTAGESTATETYPE**). If no texture is set for a stage that uses this blending argument, the system defaults to a color value of R: 1.0, G: 1.0, B: 1.0 for color, and 1.0 for alpha.

D3DTA_TFACTOR
: The texture argument is the texture factor set in a previous call to the **Direct3DDevice3.SetRenderState** with the D3DRENDERSTATE_TEXTUREFACTOR render state value.

Modifier flags

D3DTA_ALPHAREPLICATE
: Replicate the alpha information to all color channels before the operation completes.

D3DTA_COMPLEMENT
: Invert the argument such that, if the result of the argument were referred to by the variable *x*, the value would be 1.0 - *x*.

Error Codes

[This is preliminary documentation and subject to change.]

Errors, defined by the **CONST_D3DIMERR** enumeration, are represented by negative values and cannot be combined. This table lists the error codes that can be generated by all Direct3D Immediate Mode methods. See the individual method descriptions for lists of the values each can return.

D3D_OK
: No error occurred.

D3DERR_BADMAJORVERSION
: The service you requested is unavailable in this major version of DirectX. (A "major version" denotes a primary release, such as DirectX 6.0.)

D3DERR_BADMINORVERSION
: The service you requested is available in this major version of DirectX, but not in this minor version. Get the latest version of the component runtime from Microsoft. (A "minor version" denotes a secondary release, such as DirectX 6.1.)

D3DERR_COLORKEYATTACHED
: The application attempted to create a texture with a surface that uses a color key for transparency.

D3DERR_CONFLICTINGTEXTUREFILTER
: The current texture filters cannot be used together.

D3DERR_CONFLICTINGTEXTUREPALETTE
: The current textures cannot be used simultaneously. This generally occurs when a multi-texture device requires that all palettized textures simultaneously enabled also share the same palette.

D3DERR_CONFLICTINGRENDERSTATE
: The currently set render states cannot be used together.

D3DERR_DEVICEAGGREGATED
: The **IDirect3DDevice3::SetRenderTarget** method was called on a device that was retrieved from the render target surface.

D3DERR_EXECUTE_CLIPPED_FAILED
: The execute buffer could not be clipped during execution.

D3DERR_EXECUTE_CREATE_FAILED
: The execute buffer could not be created. This typically occurs when no memory is available to allocate the execute buffer.

D3DERR_EXECUTE_DESTROY_FAILED
: The memory for the execute buffer could not be deallocated.

D3DERR_EXECUTE_FAILED
: The contents of the execute buffer are invalid and cannot be executed.

D3DERR_EXECUTE_LOCK_FAILED
: The execute buffer could not be locked.

D3DERR_EXECUTE_LOCKED

The operation requested by the application could not be completed because the execute buffer is locked.

D3DERR_EXECUTE_NOT_LOCKED
The execute buffer could not be unlocked because it is not currently locked.

D3DERR_EXECUTE_UNLOCK_FAILED
The execute buffer could not be unlocked.

D3DERR_INITFAILED
A rendering device could not be created because the new device could not be initialized.

D3DERR_INBEGIN
The requested operation cannot be completed while scene rendering is taking place. Try again after the scene is completed and the **IDirect3DDevice::EndScene** method (or equivalent method) is called.

D3DERR_INVALID_DEVICE
The requested device type is not valid.

D3DERR_INVALIDCURRENTVIEWPORT
The currently selected viewport is not valid.

D3DERR_INVALIDMATRIX
The requested operation could not be completed because the combination of the currently set world, view, and projection matrices is invalid (the determinant of the combined matrix is zero).

D3DERR_INVALIDPALETTE
The palette associated with a surface is invalid.

D3DERR_INVALIDPRIMITIVETYPE
The primitive type specified by the application is invalid.

D3DERR_INVALIDRAMPTEXTURE
Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.

D3DERR_INVALIDVERTEXFORMAT
The combination of flexible vertex format flags specified by the application is not valid.

D3DERR_INVALIDVERTEXTYPE
The vertex type specified by the application is invalid.

D3DERR_LIGHT_SET_FAILED
The attempt to set lighting parameters for a light object failed.

D3DERR_LIGHTHASVIEWPORT
The requested operation failed because the light object is associated with another viewport.

D3DERR_LIGHTNOTINTHISVIEWPORT
The requested operation failed because the light object has not been associated with this viewport.

D3DERR_MATERIAL_CREATE_FAILED

The material could not be created. This typically occurs when no memory is available to allocate for the material.

D3DERR_MATERIAL_DESTROY_FAILED
The memory for the material could not be deallocated.

D3DERR_MATERIAL_GETDATA_FAILED
The material parameters could not be retrieved.

D3DERR_MATERIAL_SETDATA_FAILED
The material parameters could not be set.

D3DERR_MATRIX_CREATE_FAILED
The matrix could not be created. This can occur when no memory is available to allocate for the matrix.

D3DERR_MATRIX_DESTROY_FAILED
The memory for the matrix could not be deallocated.

D3DERR_MATRIX_GETDATA_FAILED
The matrix data could not be retrieved. This can occur when the matrix was not created by the current device.

D3DERR_MATRIX_SETDATA_FAILED
The matrix data could not be set. This can occur when the matrix was not created by the current device.

D3DERR_NOCURRENTVIEWPORT
The viewport parameters could not be retrieved because none have been set.

D3DERR_NOTINBEGIN
The requested rendering operation could not be completed because scene rendering has not begun. Call **IDirect3DDevice3::BeginScene** to begin rendering then try again.

D3DERR_NOVIEWPORTS
The requested operation failed because the device currently has no viewports associated with it.

D3DERR_SCENE_BEGIN_FAILED
Scene rendering could not begin.

D3DERR_SCENE_END_FAILED
Scene rendering could not be completed.

D3DERR_SCENE_IN_SCENE
Scene rendering could not begin because a previous scene was not completed by a call to the **IDirect3DDevice3::EndScene** method.

D3DERR_SCENE_NOT_IN_SCENE
Scene rendering could not be completed because a scene was not started by a previous call to the **IDirect3DDevice3::BeginScene** method.

D3DERR_SETVIEWPORTDATA_FAILED
The viewport parameters could not be set.

D3DERR_STENCILBUFFER_NOTPRESENT
The requested stencil buffer operation could not be completed because there is no stencil buffer attached to the render target surface.

D3DERR_SURFACENOTINVIDMEM
The device could not be created because the render target surface is not located in video-memory. (Hardware-accelerated devices require video-memory render target surfaces.)

D3DERR_TEXTURE_BADSIZE
The dimensions of a current texture are invalid. This can occur when an application attempts to use a texture that has non-power-of-two dimensions with a device that requires them.

D3DERR_TEXTURE_CREATE_FAILED
The texture handle for the texture could not be retrieved from the driver.

D3DERR_TEXTURE_DESTROY_FAILED
The device was unable to deallocate the texture memory.

D3DERR_TEXTURE_GETSURF_FAILED
The DirectDraw surface used to create the texture could not be retrieved.

D3DERR_TEXTURE_LOAD_FAILED
The texture could not be loaded.

D3DERR_TEXTURE_LOCK_FAILED
The texture could not be locked.

D3DERR_TEXTURE_LOCKED
The requested operation could not be completed because the texture surface is currently locked.

D3DERR_TEXTURE_NO_SUPPORT
The device does not support texture mapping.

D3DERR_TEXTURE_NOT_LOCKED
The requested operation could not be completed because the texture surface is not locked.

D3DERR_TEXTURE_SWAP_FAILED
The texture handles could not be swapped.

D3DERR_TEXTURE_UNLOCK_FAILED
The texture surface could not be unlocked.

D3DERR_TOOMANYOPERATIONS
The application is requesting more texture filtering operations than the device supports.

D3DERR_TOOMANYPRIMITIVES
The device is unable to render the provided quantity of primitives in a single pass.

D3DERR_UNSUPPORTEDALPHAARG
The device does not support one of the specified texture blending arguments for the alpha channel.

D3DERR_UNSUPPORTEDALPHAOPERATION
The device does not support one of the specified texture blending operations for the alpha channel.

D3DERR_UNSUPPORTEDCOLORARG

The device does not support the one of the specified texture blending arguments for color values.

D3DERR_UNSUPPORTEDCOLOROPERATION
The device does not support the one of the specified texture blending operations for color values.

D3DERR_UNSUPPORTEDFACTORVALUE
The specified texture factor value is not supported by the device.

D3DERR_UNSUPPORTEDTEXTUREFILTER
The specified texture filter is not supported by the device.

D3DERR_VBUF_CREATE_FAILED
The vertex buffer could not be created. This can happen when there is insufficient memory to allocate a vertex buffer.

D3DERR_VERTEXBUFFERLOCKED
The requested operation could not be completed because the vertex buffer is locked.

D3DERR_VERTEXBUFFEROPTIMIZED
The requested operation could not be completed because the vertex buffer is optimized. (The contents of optimized vertex buffers are driver specific, and considered private.)

D3DERR_VIEWPORTDATANOTSET
The requested operation could not be completed because viewport parameters have not yet been set. Set the viewport parameters by calling **IDirect3DViewport3::SetViewport** method and try again.

D3DERR_VIEWPORTHASNODEVICE
The requested operation could not be completed because the viewport has not yet been associated with a device. Associate the viewport with a rendering device by calling **IDirect3DDevice3::AddViewport** and try again.

D3DERR_WRONGTEXTUREFORMAT
The pixel format of the texture surface is not valid.

D3DERR_ZBUFF_NEEDS_SYSTEMMEMORY
The requested operation could not be completed because the specified device requires system-memory depth-buffer surfaces. (Software rendering devices require system-memory depth buffers.)

D3DERR_ZBUFF_NEEDS_VIDEOMEMORY
The requested operation could not be completed because the specified device requires video-memory depth-buffer surfaces. (Hardware-accelerated devices require video-memory depth buffers.)

D3DERR_ZBUFFER_NOTPRESENT
The requested operation could not be completed because the render target surface does not have an attached depth buffer.

Direct3D Immediate Mode Samples

[This is preliminary documentation and subject to change.]

This section provides summaries of the applications in the DirectX® SDK that are primarily intended to demonstrate the Direct3D® component in Immediate Mode. The following sample programs demonstrate the use and capabilities of Direct3D:

- Bend Sample
- Billboard Sample
- Boids Sample
- BumpMap Sample
- Compress Sample
- D3DFrame Library
- Filter Sample
- Fireworks Sample
- Flare Sample
- Fog Sample
- LightMap Sample
- Lights Sample
- MipMap Sample
- MTexture Sample
- PPlane Sample
- ShadowVol Sample
- ShadowVol2 Sample
- Spheremap Sample
- TunnelDP Sample
- TunnelEB Sample
- VideoTex Sample
- VBuffer Sample
- WBuffer Sample
- XFile Sample

Note

Most of the sample programs have a common interface (found in the D3DFrame Library) that has no menu bar. In addition to normal Windows® functions (minimize, maximize, restore, resize, and close), the interface supports the following commands:

Key / Input	Command

ESC	Quit
F1	Help/About Box (lists these keystrokes)
F2	Device options
ALT+ENTER	Toggle between full-screen and windowed mode
Right Mouse Button	Display options popup menu.

The device options dialog box allows the user to change the driver, Direct3D device, or display mode (full-screen only) at run time. Not all devices can support rendering in a window at all display depths. Note that 3-D hardware has finite video memory that may be exceeded for certain display modes and window sizes. In either case, the samples will display a message and default to a software rasterizer.

Although DirectX samples include Microsoft® Visual C++® project workspace files, you might need to verify other settings in your development environment to ensure that the samples compile properly. For more information, see Compiling DirectX Samples and Other DirectX Applications.

Bend Sample

[This is preliminary documentation and subject to change.]

Description

The Bend sample demonstrates a technique called surface skinning. It displays 3-D object which rotates about the y-axis and appears to bend.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Bend

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

The sample achieves the surface skinning effect by using two static copies of the object, one of which is oscillating along an axis. Each frame, the vertices of the two objects are merged and blended into a third object. The sample program displays the object derived from the third set of vertices.

This sample was built using the Direct3D sample framework.

Billboard Sample

[This is preliminary documentation and subject to change.]

Description

The Billboard sample illustrates the billboarding technique. Billboarding is a way of making 2-D sprites appear to be 3-D. It can also be used for smoke, clouds, vapor trails, energy blasts and more. For more information, see Common Techniques and Special Effects.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Billboard

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

The sample displays a grassy field with trees in it. The trees look like 3-D objects. However, they are actually 2-D texture bitmaps that are blended onto invisible rectangular polygons.

As the sample program executes, the viewpoint changes. Each time it does, all of the billboard polygons that the trees are painted onto are rotated so they face the viewer. The program then blends the images of the trees onto the billboard polygons. The trees appear to be 3-D because they can be viewed from all angles. However, close inspection reveals that the trees have exactly the same appearance from all angles. For many applications, users will not notice this minor drawback.

The shadows are also 2-D textures.

This sample was built using the Direct3D sample framework.

Boids Sample

[This is preliminary documentation and subject to change.]

Description

Boids illustrates how to write a Direct3D program. Using a flocking algorithm, the program moves a group of 3-D objects over a simple landscape.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Boids

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

The Boids program illustrates the fundamentals of creating a 3-D environment and animating a group of objects in it. From a programming perspective, the most interesting aspect is the way flocking is handled.

BumpMap Sample

[This is preliminary documentation and subject to change.]

Description

The BumpMap program demonstrates the bump mapping capabilities of Direct3D. Bump mapping is a texture blending technique used to render the appearance of rough surfaces.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Bumpmap

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Your graphics hardware might not support bump mapping, in which case Direct3D displays a message to that effect when you attempt to run this program. The solution is to enable the reference rasterizer. You can do so by running *(SDK root)* \Samples\Multimedia\D3dim\Bin\Enablerefrast.reg.

Programming Notes

Bump mapping is an advanced multitexture blending technique that can be used to render the appearance of rough surfaces. The bump map itself is a texture that stores the perturbation data.

In this sample program, the map of the world is a texture. The program blends both the map texture and the bump map texture onto the sphere to give the appearance of a high-resolution topographical map.

For more details on this technique, see Bump Mapping.

This sample was built using the Direct3D sample framework.

Compress Sample

[This is preliminary documentation and subject to change.]

Description

The Compress sample demonstrates how to load the DDS file format into a compressed texture surface. DDS textures can be created using the DxTex program included with the DirectX SDK.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Compress

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

The file format is called DDS because it encapsulates the information in a DirectDrawSurface. The data can be read directly into a surface of a matching format.

The ReadDDSTexture function demonstrations how a DDS surface is read from a file.

A DDS file has the following format:

DWORD dwMagic	(0x20534444, or "DDS ")
DDSURFACEDESC2 ddsd	Information about the surface format
BYTE bData1[]	Data for the main surface
[BYTE bData2[]	Data for attached surfaces, if any, follow

This format is easy to read and write, and supports features such as alpha and multiple mip levels, as well as DXTn compression. If it uses DXTn compression, it may be one of 5 compressed types. See Compressed Texture Surfaces.

After the texture is read in, a pixel format must be chosen that is supported by the renderer. In the sample, the supported pixel formats are enumerated and stored in a

linked list. After the pixel formats are collected, the list is searched for a best match, using the FindBestPixelFormatMatch function.

Some Direct3D devices such as the reference rasterizer and some hardware devices can render compressed textures directly. But for renderers that don't directly support this, the compressed surface must be blitted to a non-compressed surface. The function BltToUncompressedSurface demonstrates how this is done.

D3DFrame Library

[This is preliminary documentation and subject to change.]

Description

The D3D framework is a set of C++ classes that were used to create the Direct3D Immediate Mode sample programs. It is not a part of the Direct3D API. The framework was created and used to provide consistency and clarity of presentation for the Direct3D samples. The framework classes may or may not be appropriate for use in your applications.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\D3DFrame

Executable: None.

User's Guide

D3DFrame compiles as a static linker library, which is used to build the remainder of the Direct3D Immediate Mode samples.

Programming Notes

The framework consists of classes that enumerate the DirectDraw drivers, Direct3D devices, and display modes available to each device. The sample programs use them to initialize and run Direct3D. The classes also help provide a consistent user interface for the set of sample programs. In addition, the framework includes a set of classes for loading and managing textures.

The Direct3D framework also contains numerous macros and functions for debugging, for manipulating Direct3D objects, and for doing math operations common in Direct3D programming.

Filter Sample

[This is preliminary documentation and subject to change.]

Description

The Filter sample program demonstrates the texture filtering techniques that Direct3D supports. Direct3D texture filtering techniques enable applications to achieve a greater realism in the appearance of rendered primitives. For more information, see Texture Filtering.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Filter

Executable: None.

User's Guide

When the program begins, it displays two rectangular primitives with textures on them.

In addition to the usual commands listed in the **About** box when you press F1, this program has a main menu.

The **File** menu contains choices for pausing and resuming the program, changing the Direct3D device, and exiting the program. You can also pause and resume the application by pressing the ENTER key on your keyboard.

The **Left Pane** menu controls the texture filtering methods that the program uses for magnification and minification of the 3-D primitive on the left side of the screen. It also has options for edge antialiasing and anisotropic texture filtering. You may have to enable to software reference rasterizer to view the effects of these options. To enable the reference rasterizer, run *(SDK root)* \Samples\Multimedia\D3dim\Bin\Enablerefrast.reg.

The **Right Pane** menu enables you to set the filtering methods that the program uses to perform magnification and minification when it renders the primitive on the right side of the screen.

Programming Notes

This program demonstrates nearest point sampling, linear filtering, and anisotropic texture filtering. It illustrates how to enable and disable anisotropy. In addition, this sample shows how your program can set Direct3D to perform edge antialiasing.

Fireworks Sample

[This is preliminary documentation and subject to change.]

Description

Fireworks implements a system of particles simulating a fireworks explosion. Particles are popular in games to show effects like smoke, sparks, and explosions.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Fireworks

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

The fireworks explosion is simulated by using a system of particles, where each particle is rendered with a partially transparent texture map of a sphere. The position and color of each particle are governed by parameterized equations and are updated in each frame.

Flare Sample

[This is preliminary documentation and subject to change.]

Description

The Flare sample shows how to create a lens flare effect using alpha blending.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Flare

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

In this sample, lens flare is simulated with mathematical functions which govern the position of each flare. The flares are rendered using additive alpha blending, and animated to give a sparkle effect.

Fog Sample

[This is preliminary documentation and subject to change.]

Description

The Fog sample does a fly-by over some terrain with fog enabled.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Fog

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands. The effects are most striking in full-screen mode.

Programming Notes

Games that use terrain typically turn on fog so they can prevent the rendering of objects in the far distance. It looks like a cool effect, but is actually used to get high performance.

LightMap Sample

[This is preliminary documentation and subject to change.]

Description

This sample shows how to use multitexturing and multipass techniques to do some complex lighting effects. There is a light swinging in a room, which dynamically lights up the walls and ceiling as the light moves.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\LightMap

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

On the **Options** menu, choose between multipass and multiple texture blending. The latter will not be available if your hardware does not support it.

Press F1 to see other available commands.

Programming Notes

There is no “true” lighting in this sample–everything is done with light maps. Light maps are extremely popular in games theses days, because they are much faster than real lighting. Also, real lighting is calculated only at the vertices, so highly tesselated meshes are required.

Lights Sample

[This is preliminary documentation and subject to change.]

Description

The Lights sample shows how to turn on and move the various types of Direct3D lights. The lights fly around, illuminating the scene, and switch every few seconds to a new light type.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Lights

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

This sample shows how to set up the **D3DLIGHT** structure for each of the several types of lights available. It also shows how to dynamically orient the lights on each frame.

MipMap Sample

[This is preliminary documentation and subject to change.]

Description

This sample shows two brick walls moving back and forth along the z-axis. One uses mipmapping and the other does not, giving a side-by-side comparison of the benefits of mipmapping.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\MipMap

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

This sample does not use the texture code of D3DFrame, but actually shows the full implementation needed to load and build mipmapped textures.

MTexture Sample

[This is preliminary documentation and subject to change.]

Description

The MTexture program shows how to use multitexturing. The scene consists of a room with walls that have a base texture and a spotlight texture, each using a different set of texture coordinates.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\MTexture

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

This sample shows how to program the multitexture stages using the new **IDirect3DDevice3::SetTextureStageState** method. Dozens of different effects are attainable with this method. The sample just shows one, monochrome light mapping, which is very popular in current game titles.

PPlane Sample

[This is preliminary documentation and subject to change.]

Description

The PPlane program is a simple example of how to create Direct3D Immediate Mode programs.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Pplane

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

In addition to demonstrating Direct3D Immediate Mode programming techniques, this application contains a flocking algorithm used to animate the paper planes.

ShadowVol Sample

[This is preliminary documentation and subject to change.]

Description

The ShadowVol sample demonstrates how to create and use stencil buffers to implement shadow volumes. With shadow volumes, an arbitrarily shaped object can cast a shadow onto another arbitrarily shaped object.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\ShadowVol

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

This sample will run only on devices that support stencil buffers.

Press F1 to see available commands.

Programming Notes

Shadow volumes are a fairly advanced technique. To start, take an object that you'd like to have cast a shadow. From that object, build a set of polygonal faces that encompass the volume of its shadow. Next, use the stencil buffer to render the front facing planes of the shadow volume. Then, set up the stencil buffer to render the back-facing planes, this time subtracting values from the stencil buffer. Afterwards, the stencil buffer contains a mask of the cast shadow. Just draw a large gray or black rectangle using the stencil buffer as a mask, and the frame buffer will get updated with the shadow.

ShadowVol2 Sample

[This is preliminary documentation and subject to change.]

Description

The ShadowVol2 sample demonstrates how to create and use stencil buffers to implement shadow volumes, which are used to cast shadows on arbitrarily complex objects. It is an extension of the ShadowVol Sample.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\ShadowVol2

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

This sample will run only on devices that support stencil buffers.

Press F1 to see available commands.

The following options are available on the **Shadow Modes** menu:

- **Draw Shadows**: Check this option to enable shadow rendering.
- **Show Shadow Volumes**: Instead of shadows, draw the shadow volumes used to compute them.
- **Draw Shadow Volume Caps**: If this is turned off, "extra" shadows may be visible where the far caps of the directional-light cylindrical shadow volumes happen to be visible.
- **Show Quarter Viewpoint**: Show scene from a different angle.
- **1 Bit Stencil Mode**: Use different algorithm that uses only 1 bit of stencil buffer, where overlapping shadows are not allowed. If the device only supports 1-bit stencil, you will not be allowed to switch out of this mode.
- **Z Order Shadow Vols**: In 1-Bit Mode, shadow volumes must be rendered front-to-back, so rendering may be incorrect unless this option is checked.

Programming Notes

Shadow volumes are a technique for casting shadows onto arbitrary non-planar surfaces. The effect is achieved by constructing a shadow volume with respect to the light source and the shadow caster. In this example, the light source is a directional light whose direction circles about points on the plane, and the shadow volume is computed by projecting the vertices of the shadow caster onto a plane perpendicular to the light, finding the 2-D convex hull of these points in the plane, and extruding the 2-D hull in the light direction to form the 3-D shadow volume. The shadow volume must extend far enough so that it covers any geometry that will be in shadow. This particular shadow volume computation requires that the shadow caster be a convex object.

The rendering proceeds as follows. First the geometry is rendered as normal, then the shadow volume is rendered without writing to the z or color buffer (alpha blending is

used here to avoid writes to the color buffer). Every place the shadow volume appears is marked in the stencil buffer. Next, the cull order is reversed and the back faces of the shadow volume are rendered, this time unmarking all the pixels that are covered in the stencil buffer. These have passed the z-test, and thus are visible behind the back of the shadow volume, so they are not in shadow. The pixels still marked are those that lie inside the front and back bounds of the shadow volume and are thus in shadow. These pixels are blended with a large black rectangle that covers the viewport, generating the shadow.

Spheremap Sample

[This is preliminary documentation and subject to change.]

Description

This samples loads a 3-D object and renders it using a sphere map.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Spheremap

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

The sphere map itself is a special, preconstructed texture map containing a 180-degree view of an environment. Before a frame is rendered, the object's normals are used to compute the texture coordinates for each vertex of the object. When rendered, the object looks as if it reflects the environment.

TunnelDP Sample

[This is preliminary documentation and subject to change.]

Description

This sample uses **DrawPrimitive** to render objects.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\TunnelDP

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands. In addition, you can change some effects by using the main menu.

Programming Notes

The point of this sample is to compare its source code to that of TunnelEB, the identical version using execute buffers.

TunnelEB Sample

[This is preliminary documentation and subject to change.]

Description

This sample uses execute buffers to render objects.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\TunnelEB

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands. In addition, you can change some effects by using the main menu.

Programming Notes

The point of this sample is to compare its source code to that of TunnelDP, the identical version using **DrawPrimitive**.

VBuffer Sample

[This is preliminary documentation and subject to change.]

Description

This sample shows how to use the vertex buffers that are new to DirectX 6.0.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\VBuffer

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see available commands.

Programming Notes

Before vertex buffers, geometry rendering was accomplished by using an array of vertices passed to **DrawPrimitive** calls. Vertex buffers are more useful, but need to be specifically created, locked, filled, and unlocked before being rendered with the new **DrawPrimitiveVB** calls.

VideoTex Sample

[This is preliminary documentation and subject to change.]

The VideoTex sample shows how to use an .avi file as a texture map.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\VideoTex

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see a list of the usual commands.

Programming Notes

The program draws a cube with an .avi texture mapped to each of its faces. The key is to have the texture's surface use the DDSCAPS2_HINTDYNAMIC flag.

WBuffer Sample

[This is preliminary documentation and subject to change.]

Description

The WBuffer sample shows how to use w-buffering.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\WBuffer

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

Press F1 to see a list of the usual commands. In addition, you can change the buffering mechanism by pressing W (w-buffering, if supported), Z (z-buffering), or N (no buffering). Note the artifacts that appear when using z-buffering.

Programming Notes

W-buffering is a depth-buffering alternative to z-buffering, and should be used in cases where z-buffering produces artifacts. W-buffering does a much better job of quantizing the depth buffer.

XFile Sample

[This is preliminary documentation and subject to change.]

Description

This sample shows how to load and render .x files.

Path

Source: *(SDK root)*\Samples\Multimedia\D3dim\Src\Xfile

Executable: *(SDK root)*\Samples\Multimedia\D3dim\Bin

User's Guide

When you run the program, an **Open File** dialog box appears. You can find some .x files in the media folder of the D3DIM samples directory. If there is a .bmp file of the same name in the directory, the bmp is automatically used as a texture for the object.

Press F1 to see available commands. In addition, you can load a different file from the **File** menu.

Programming Notes

If a texture appears upside-down on an object you have loaded, it is because of the way the object loads the texture. An earlier version of Direct3D loaded bitmaps upside-down, so the creators of some .x files compensated by reversing the coordinates.